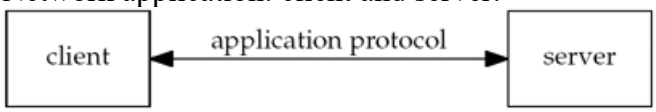
Network Programming

Unit 1

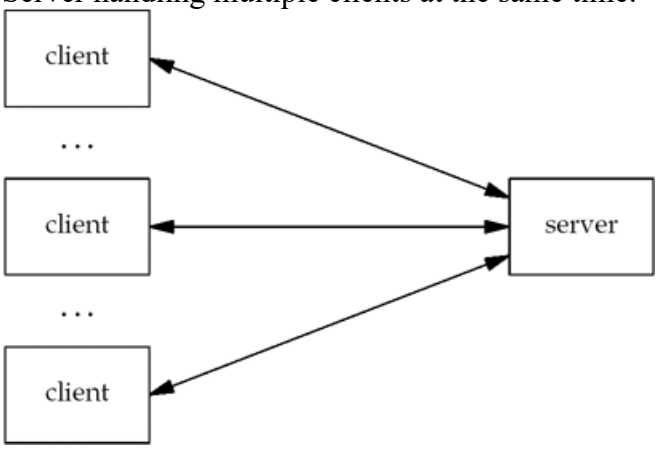
Q. What is network protocol? With a neat block diagram explain the network application for client and server.

- A network protocol is an established set of rules that determine how data is transmitted between different devices in the same network.
- Essentially, it allows connected devices to communicate with each other, regardless of any differences in their internal processes, structure or design.

Network application: client and server:



Server handling multiple clients at the same time:



(When writing programs that communicate across a computer network, one must first invent a protocol, an agreement on how those programs will communicate. Before delving into the design details of a protocol, high-level decisions must be made about which program is expected to initiate communication and when responses are expected. For example, a Web server is typically thought of as a long-running program (or daemon) that sends network messages only in response to requests coming in from the network. The other side of the protocol is a Web client, such as a browser, which always initiates communication with the server.)

- Clients normally communicate with one server at a time, although using a Web browser as an example, we might communicate with many different Web servers.
- But from the server's perspective, at any given point in time, it is not unusual for a server to be communicating with multiple clients.
- The client application and the server application may be thought of as communicating via a network protocol, but actually, multiple layers of network protocols are typically involved.

Q.List out the various approaches used to handle multiple clients at the same time.

Server can handle multiple clients by :

1. Multi threading
2. Socket Programming

Multi threading :

The simple way to handle multiple clients would be to spawn a new thread for every new client connected to the server.

Semaphore is simply a variable that is non-negative and shared between threads. This variable is used to solve the critical section problem and to achieve process synchronization in the multiprocessing environment.

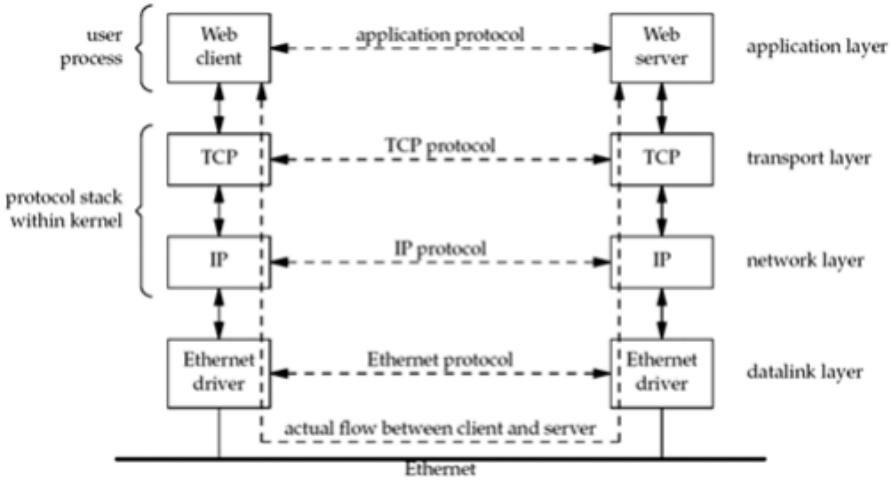
Socket Programming:

This method is strongly not recommended because of the following disadvantages:

- Threads are difficult to code, debug and sometimes they have unpredictable results.
- Overhead switching of context
- Not scalable for large number of clients
- Deadlocks can occur

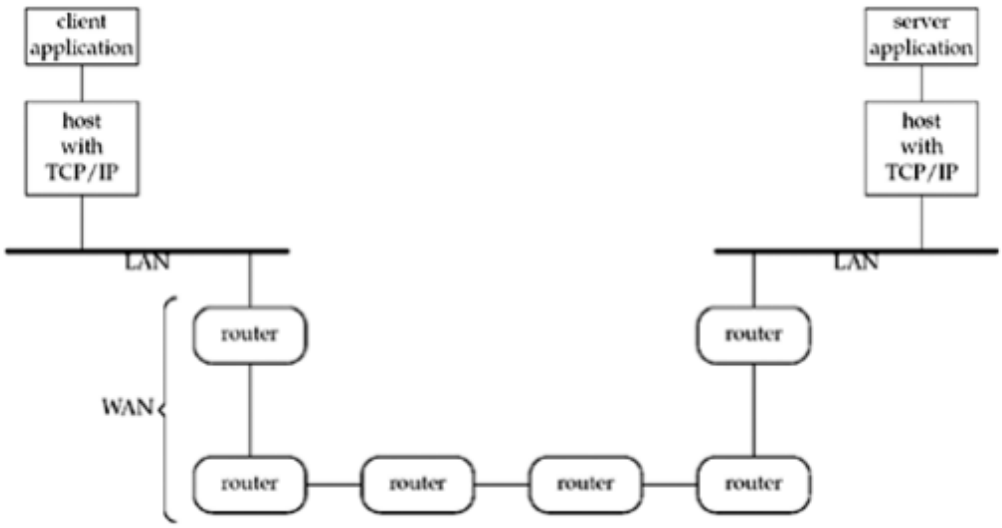
When you design a server program, plan for multiple concurrent processes. Special socket calls are available for that purpose they are called *concurrent servers*, as opposed to the more simple type of *iterative server*

Q.With a neat block diagram, explain the client and server communication on Local Area Network using TCP.



- Even though the client and server communicate using an application protocol, the transport layers communicate using TCP.
- The actual flow of information between client and server goes down the protocol stack on one side, across the network, and up the protocol stack on the other side.
- Client & Server are typically user processes, while TCP and IP protocols are normally part of the protocol stack within the kernel.
- The four layers labeled in the diagram are the Application layer, Transport layer, Network layer, Data-link layer.
- Some clients and servers use the User Datagram Protocol (UDP) instead of TCP.

Q.With a neat block diagram, explain the client and server communication over Wide Area Network using TCP.



- The client & server on different LANs is connected to a Wide Area Network (WAN) via a router.
- Routers are the building blocks of WANs.
- The largest WAN today is the Internet.
- Many companies build their own WANs and these private WANs may or may not be connected to the Internet.

Q.List and explain the steps involved in simple daytime client.

1. **Include headers** :
   Headers includes numerous system headers that are needed by most network programs and defines various constants that we use .

2. **Command-line arguments:**
   Definition of the main function along with the command-line arguments.

3. **Create TCP socket :**
   The socket function creates an Internet (AF_INET) stream (SOCK_STREAM) socket, which is a fancy name for a TCP socket. The function returns a small integer descriptor that we can use to identify the socket in all future function calls.
   The if statement contains a call to the socket function, an assignment of the return value to the variable named sockfd, and then a test of whether this assigned value is less than 0. While we could break this into two C statements,
   sockfd = socket(AF_INET, SOCK_STREAM, 0);
   if (sockfd < 0)
   If the call to socket fails, we abort the program by calling our own err_sys function. It prints our error message along with a description of the system error that occurred (e.g., "Protocol not supported" is one possible error from socket) and terminates the process.

4. **Specify server's IP address and port :**

Fill in an Internet socket address structure (sockaddr_in) with the server's IP address and port number.
We set the entire structure to 0 using bzero, set the address family to AF_INET, set the port number to 13, and set the IP address to the value specified as the first command-line argument (argv[1]).

5. **Establish connection with server:**
   The connect function, when applied to a TCP socket, establishes a TCP connection with the server specified by the socket address structure pointed to by the second argument. We must also specify the length of the socket address structure as the third argument to connect, and for Internet socket address structures, we always let the compiler calculate the length using C's sizeof operator.

6. **Read and display server's reply:**
   We read the server's reply and display the result using the standard I/O functions.
   We must be careful when using TCP because it is a byte-stream protocol with no record boundaries. The server's reply is normally a 26-byte string.
   With a byte-stream protocol, these 26 bytes can be returned in numerous ways: a single TCP segment containing all 26 bytes of data, in 26 TCP segments each containing 1 byte of data, or any other combination that totals to 26 bytes

7. **Terminate program:**
   exit terminates the program. Unix always closes all open descriptors when a process terminates, so our TCP socket is now closed.

Q.Develop a 'C' program for simple daytime client

Figure 1.5 TCP daytime client.

intro/daytimetcpcli.c

```
 1 #include  "unp.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     sockfd, n;
 6     char    recvline[MAXLINE + 1];
 7     struct sockaddr_in servaddr;

 8     if (argc != 2)
 9         err_quit("usage: a.out <IPaddress>");

10     if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
11         err_sys("socket error");

12     bzero(&servaddr, sizeof(servaddr));
13     servaddr.sin_family = AF_INET;
14     servaddr.sin_port = htons(13);  /* daytime server */
15     if (inet_pton(AF_INET, argv[1], &servaddr.sin_addr) <= 0)
16         err_quit("inet_pton error for %s", argv[1]);

17     if (connect(sockfd, (SA *) &servaddr, sizeof(servaddr)) < 0)
18         err_sys("connect error");

19     while ( (n = read(sockfd, recvline, MAXLINE)) > 0) {
20         recvline[n] = 0;        /* null terminate */
21         if (fputs(recvline, stdout) == EOF)
22             err_sys("fputs error");
23     }
24     if (n < 0)
25         err_sys("read error");

26     exit(0);
27 }
```

Q.Protocol Independence

- It is better to make a program *protocol-independent*.
- Protocol-independent is achieved by using the `getaddrinfo` function (which is called by `tcp_connect`).
- Another deficiency in our programs is that the user must enter the server's IP address as a dotted-decimal number. Humans work better with names instead of numbers.

Modify the day time client program for IPv6.

```
#include     "unp.h"

int
main(int argc, char **argv)
{
    int     sockfd, n;
    char    recvline[MAXLINE + 1];
    struct sockaddr_in6 servaddr;

    if (argc != 2)
        err_quit("usage: a.out <IPaddress>");

    if ( (sockfd = socket(AF_INET6, SOCK_STREAM, 0)) < 0)
        err_sys("socket error");

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin6_family = AF_INET6;
    servaddr.sin6_port = htons(13);    /* daytime server */
    if (inet_pton(AF_INET6, argv[1], &servaddr.sin6_addr) <= 0)
        err_quit("inet_pton error for %s", argv[1]);

    if (connect(sockfd, (SA *) &servaddr, sizeof(servaddr)) < 0)
        err_sys("connect error");

    while ( (n = read(sockfd, recvline, MAXLINE)) > 0) {
        recvline[n] = 0;    /* null terminate */
        if (fputs(recvline, stdout) == EOF)
            err_sys("fputs error");
    }
    if (n < 0)
        err_sys("read error");

    exit(0);
}
```

Q.What are wrapper functions? Develop the wrapper function for the following:

a. **Socket function**
b. **Pthread_mutex_lock**

Since terminating on an error is the common case, we can shorten our programs by defining a *wrapper function* that performs the actual function call, tests the return value, and terminates on an error. The convention we use is to capitalize the name of the function.
Detailed answer:
In any real-world program, it is essential to check every function call for an error return. We check for errors from socket, inet_pton, connect, read, and fputs, and when one occurs, we call our own functions,
err_quit and err_sys, to print an error message and terminate the program.
We find that most of the time, this is what we want to do. Occasionally, we want to do something other than terminate when one of these functions returns an error, when we must check for an interrupted system call. Since terminating on an error is the common case, we can shorten our programs by defining a wrapper function that performs the actual function call, tests the return value, and terminates on an error. The convention we use is to capitalize the name of the function, as in
sockfd = Socket(AF_INET, SOCK_STREAM, 0);

a)Socket function:

```
int Socket(int family, int type, int protocol)
{
        int n;
        if ( (n = socket(family, type, protocol)) < 0)
                err_sys("socket error");
        return (n);
}
```

b)Pthread_mutex_lock:

```
Void Pthread_mutex_lock(pthread_mutex_t *mptr)
{
        int n;
        if ( (n = pthread_mutex_lock(mptr)) == 0)
        return;
        errno = n;
        err_sys("pthread_mutex_lock error");
}
```

Q.List and explain the steps involved in simple daytime server

1. Create a TCP socket:
   The creation of the TCP socket is identical to the client code.

2. Bind server's well-known port to socket:
   The server's well-known port (13 for the daytime service) is bound to the socket by filling in an Internet socket address structure and calling bind. We specify the IP address as INADDR_ANY, which allows the server to accept a client connection on any interface, in case the server host has multiple interfaces. Later we will see how we can restrict the server to accepting a client connection on just a single interface.

3. Convert socket to listening socket:
   By calling listen, the socket is converted into a listening socket, on which incoming connections from clients will be accepted by the kernel. These three steps, socket, bind, and listen, are the normal steps for any TCP server to prepare what we call the listening descriptor (listenfd in this example). The constant LISTENQ is from our unp.h header. It specifies the maximum number of client connections that the kernel will queue for this listening descriptor.

4. Accept client connection, send reply:
   Normally, the server process is put to sleep in the call to accept, waiting for a client connection to arrive and be accepted. A TCP connection uses what is called a three-way handshake to establish a connection. When this handshake completes, accept returns, and the return value from the function is a new descriptor (connfd) that is called the connected descriptor. This new descriptor is used for communication with the new client. A new descriptor is returned by accept for each client that connects to our server.

5. Terminate connection:
   The server closes its connection with the client by calling close. This initiates the normal TCP connection termination sequence: a FIN is sent in each direction and each FIN is acknowledged by the other end.

Q.Develop the 'C' program to implement simple daytime server

Figure 1.9 TCP daytime server.

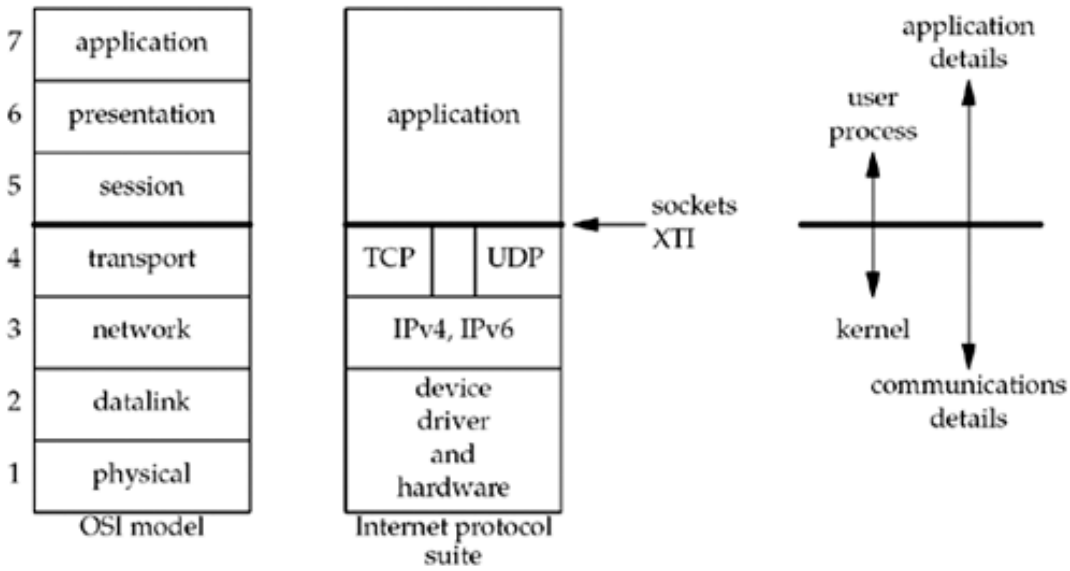
intro/daytimetcpsrv.c

```
 1 #include     "unp.h".
 2 #include     <time.h>

 3 int
 4 main(int argc, char **argv)
 5 {
 6     int     listenfd, connfd;
 7     struct sockaddr_in servaddr;
 8     char    buff[MAXLINE];
 9     time_t ticks;

10     listenfd = Socket(AF_INET, SOCK_STREAM, 0);

11     bzeros(&servaddr, sizeof(servaddr));
12     servaddr.sin_family = AF_INET;
13     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
14     servaddr.sin_port = htons(13); /* daytime server */

15     Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

16     Listen(listenfd, LISTENQ);

17     for ( ; ; ) {
18         connfd = Accept(listenfd, (SA *) NULL, NULL);

19         ticks = time(NULL);
20         snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
21         Write(connfd, buff, strlen(buff));

22         Close(connfd);
23     }
24 }
```

Q.Write a note on Unix errno value.

- When an error occurs in a Unix function (such as one of the socket functions), the global variable errno is set to a positive value indicating the type of error
- err_sys function looks at the value of errno and prints the corresponding error message string (e.g., "Connection timed out" if errno equals ETIMEDOUT)
- The value of errno is set by a function only if an error occurs.
- Its value is undefined if the function does not return an error.
- All of the positive error values are constants with all-uppercase names beginning with "E," and are normally defined in the <sys/errno.h> header. No error has a value of 0.

Q.Explain with a neat block diagram the layers of OSI model and Internet protocol suite.



A common way to describe the layers in a network is to use the International Organization for Standardization (ISO) *open systems interconnection* (OSI) model for computer communications. This is a seven-layer model,

We consider the bottom two layers of the OSI model(Datalink+ physical) as the device driver and networking hardware that are supplied with the system. Normally, we need not concern ourselves with these layers other than being aware of some properties of the datalink

The network layer is handled by the IPv4 and IPv6 protocols

We show a gap between TCP and UDP to indicate that it is possible for an application to bypass the transport layer and use IPv4 or IPv6 directly. This is called a *raw socket.*

The upper three layers of the OSI model are combined into a single layer called the application. This is the Web client (browser), Telnet client, Web server, FTP server, or whatever application we are using.

Q.Write a note on 64- bit architectures.

During the mid to late 1990s, the trend began toward 64-bit architectures and 64-bit software. One reason is for larger addressing within a process (i.e., 64-bit pointers), which can address large amounts of memory (more than 232 bytes).

The common programming model for existing 32-bit Unix systems is called the ILP32 model, denoting that integers (I), long integers (L), and pointers (P) occupy 32 bits. The model that is becoming most prevalent for 64-bit Unix systems is called the LP64 model, meaning only long integers (L) and pointers (P) require 64 bits. Figure 1.17 compares these two models.

Comparison of number of bits to hold various datatypes for the ILP32 and LP64 models:

Datatype	ILP32 model	LP64 model
char	8	8
short	16	16
int	32	32
long	32	64
pointer	32	64

Q. Explain the features of the following protocols:

a. **IPv4**
b. **IPv6**
c. **TCP**
d. **UDP**
e. **SCTP**
f. **ICMP**
g. **IGMP**
h. **ARP**
i. **RARP**
j. **ICMPv6**
k. **BPF**
l. **DLPI**

a)IPv4 Internet Protocol version 4.
IPv4, which we often denote as just IP, has been the workhorse protocol of the IP suite since the early 1980s. It uses 32-bit addresses (Section A.4). IPv4 provides packet delivery service for TCP, UDP, SCTP, ICMP, and IGMP.

b)IPv6 Internet Protocol version 6.
IPv6 was designed in the mid-1990s as a replacement for IPv4. The major change is a larger address comprising 128 bits (Section A.5), to deal with the explosive growth of the Internet in the 1990s. IPv6 provides packet delivery service for TCP, UDP, SCTP, and ICMPv6.

c)TCP Transmission Control Protocol.
TCP is a connection-oriented protocol that provides a reliable, full-duplex byte stream to its users. TCP sockets are an example of stream sockets. TCP takes care of details such as acknowledgments, timeouts, retransmissions, and the like. Most Internet application programs use TCP. Notice that TCP can use either IPv4 or IPv6.

d)UDP User Datagram Protocol.
UDP is a connectionless protocol, and UDP sockets are an example of datagram sockets. There is no guarantee that UDP datagrams ever reach their intended destination. As with TCP, UDP can use either IPv4 or IPv6.

e)SCTP Stream Control Transmission Protocol.

SCTP is a connection-oriented protocol that provides a reliable full-duplex association. The word "association" is used when referring to a connection in SCTP because SCTP is multihomed, involving a set of IP addresses and a single port for each side of an association. SCTP provides a message service, which maintains record boundaries. As with TCP and UDP, SCTP can use either IPv4 or IPv6, but it can also use both IPv4 and IPv6 simultaneously on the same association.

f)ICMP Internet Control Message Protocol.
ICMP handles error and control information between routers and hosts. These messages are normally generated by and processed by the TCP/IP networking software itself, not user processes, although we show the ping and traceroute programs, which use ICMP. We sometimes refer to this protocol as ICMPv4 to distinguish it from ICMPv6.

g)IGMP Internet Group Management Protocol.
IGMP is used with multicasting (Chapter 21), which is optional with IPv4.

h)ARP Address Resolution Protocol.
ARP maps an IPv4 address into a hardware address (such as an Ethernet address). ARP is normally used on broadcast networks such as Ethernet, token ring, and FDDI, and is not needed on point-to-point networks.

i)RARP Reverse Address Resolution Protocol.
RARP maps a hardware address into an IPv4 address. It is sometimes used when a diskless node is booting.

j)ICMPv6 Internet Control Message Protocol version 6.
ICMPv6 combines the functionality of ICMPv4, IGMP, and ARP.

k)BPF BSD packet filter.
This interface provides access to the datalink layer. It is normally found on Berkeley-derived kernels.

l)DLPI Datalink provider interface.
This interface also provides access to the datalink layer. It is normally provided with SVR4.

Q.List and explain the features of TCP Protocol in detail.

1. TCP is reliable protocol. That is, the receiver always sends either positive or negative acknowledgement about the data packet to the sender, so that the sender always has bright clue about whether the data packet is reached the destination or it needs to resend it.

2. TCP ensures that the data reaches intended destination in the same order it was sent.

3. TCP is connection oriented. TCP requires that connection between two remote points be established before sending actual data.

4. TCP provides error-checking and recovery mechanism.

5. TCP provides end-to-end communication.
6. TCP operates in Client/Server point-to-point mode.
7. TCP provides full duplex server

Q.List and explain the features of UDP Protocol in detail.

1. UDP is used when acknowledgement of data does not hold any significance.
2. UDP is good protocol for data flowing in one direction.
3. UDP is simple and suitable for query based communications.
4. UDP is not connection oriented.
5. UDP does not provide congestion control mechanism.
6. UDP does not guarantee ordered delivery of data.
7. UDP is stateless.
8. UDP is suitable protocol for streaming applications such as VoIP, multimedia streaming.

(TEXTBOOK ANSWER)
User Datagram Protocol (UDP)

- UDP is a simple transport-layer protocol. It is described in RFC 768 [Postel 1980]. The application writes a message to a UDP socket, which is then encapsulated in a UDP datagram, which is then further encapsulated as an IP datagram, which is then sent to its destination. There is no guarantee that a UDP datagram will ever reach its final destination, that order will be preserved across the network, or that datagrams arrive only once.
- The problem that we encounter with network programming using UDP is its lack of reliability. If a datagram reaches its final destination but the checksum detects an error, or if the datagram is dropped in the network, it is not delivered to the UDP socket and is not automatically retransmitted. If we want to be certain that a datagram reaches its destination, we can build lots of features into our application: acknowledgments from the other end, timeouts, retransmissions, and the like.
- Each UDP datagram has a length. The length of a datagram is passed to the receiving application along with the data. We have already mentioned that TCP is a byte-stream protocol, without any record boundaries at all which differs from UDP.
- We also say that UDP provides a connectionless service, as there need not be any long-term relationship between a UDP client and server. For example, a UDP client can create a socket and send a datagram to a given server and then immediately send another datagram on the same socket to a different server. Similarly, a UDP

server can receive several datagrams on a single UDP socket, each from a different client.

Transmission Control Protocol (TCP)

- The service provided by TCP to an application is different from the service provided by UDP. TCP is described in RFC 793 [Postel 1981c], and updated by RFC 1323 [Jacobson, Braden, and Borman 1992], RFC 2581 [Allman, Paxson, and Stevens 1999], RFC 2988 [Paxson and Allman 2000], and RFC 3390 [Allman, Floyd, and Partridge 2002]. First, TCP provides connections between clients and servers. A TCP client establishes a connection with a given server, exchanges data with that server across the connection, and then terminates the connection.
- TCP also provides reliability. When TCP sends data to the other end, it requires an acknowledgment in return. If an acknowledgment is not received, TCP automatically retransmits the data and waits a longer amount of time. After some number of retransmissions, TCP will give up, with the total amount of time spent trying to send data typically between 4 and 10 minutes (depending on the implementation).
- Note that TCP does not guarantee that the data will be received by the other endpoint, as this is impossible. It delivers data to the other endpoint if possible, and notifies the user (by giving up on retransmissions and breaking the connection) if it is not possible. Therefore, TCP cannot be described as a 100% reliable protocol; it provides reliable delivery of data or reliable notification of failure.
- TCP contains algorithms to estimate the round-trip time (RTT) between a client and server dynamically so that it knows how long to wait for an acknowledgment. For example, the RTT on a LAN can be milliseconds while across a WAN, it can be seconds. Furthermore, TCP continuously estimates the RTT of a given connection, because the RTT is affected by variations in the network traffic.
- TCP also sequences the data by associating a sequence number with every byte that it sends. For example, assume an application writes 2,048 bytes to a TCP socket, causing TCP to send two segments, the first containing the data with sequence numbers 1 1,024 and the second containing the data with sequence numbers 1,025 2,048. (A segment is the unit of data that TCP passes to IP.) If the segments arrive out of order, the receiving TCP will reorder the two segments based on their sequence numbers before passing the data to the receiving application. If TCP receives duplicate data from its peer (say the peer thought a segment was lost and retransmitted it, when it wasn't really lost, the network was just overloaded), it can detect that the data has been duplicated (from the sequence numbers), and discard the duplicate data.
- There is no reliability provided by UDP. UDP itself does not provide anything like acknowledgments, sequence numbers, RTT estimation, timeouts, or retransmissions. If a UDP datagram is duplicated in the network, two copies can be delivered to the receiving host. Also, if a UDP client sends two datagrams to the same destination, they can be reordered by the network and arrive out of order. UDP applications must handle all these cases.
- TCP provides flow control. TCP always tells its peer exactly how many bytes of data it is willing to accept from the peer at any one time. This is called the advertised window. At any time, the window is the amount of room currently available in the receive buffer, guaranteeing that the sender cannot overflow the receive buffer. The

window changes dynamically over time: As data is received from the sender, the window size decreases, but as the receiving application reads data from the buffer, the window size increases. It is possible for the window to reach 0: when TCP's receive buffer for a socket is full and it must wait for the application to read data from the buffer before it can take any more data from the peer.

- UDP provides no flow control. It is easy for a fast UDP sender to transmit datagrams at a rate that the UDP receiver cannot keep up with.
- Finally, a TCP connection is full-duplex. This means that an application can send and receive data in both directions on a given connection at any time. This means that TCP must keep track of state information such as sequence numbers and window sizes for each direction of data flow: sending and receiving. After a full-duplex connectio

Feature	TCP	UDP
Connection status	Requires an established connection to transmit data (connection should be closed once transmission is complete)	Connectionless protocol with no requirements for opening, maintaining, or terminating a connection
Data sequencing	Able to sequence	Unable to sequence
Guaranteed delivery	Can guarantee delivery of data to the destination router	Cannot guarantee delivery of data to the destination
Retransmission of data	Retransmission of lost packets is possible	No retransmission of lost packets
Error checking	Extensive error checking and acknowledgment of data	Basic error checking mechanism using checksums
Method of transfer	Data is read as a byte stream; messages are transmitted to segment boundaries	UDP packets with defined boundaries; sent individually and checked for integrity on arrival
Speed	Slower than UDP	Faster than TCP
Broadcasting	Does not support Broadcasting	Does support Broadcasting
Optimal use	Used by HTTPS, HTTP, SMTP, POP, FTP, etc	Video conferencing, streaming, DNS, VoIP, etc

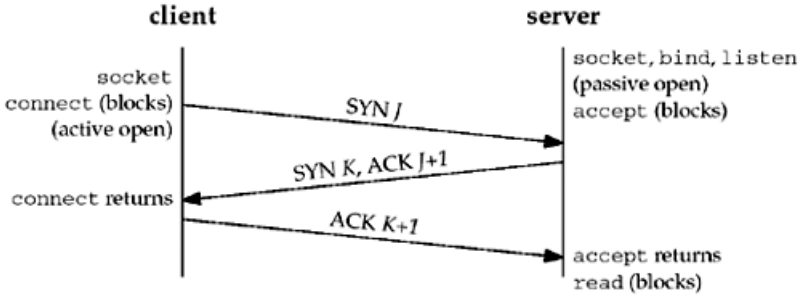
Q.Explain with a neat diagrams the following:
a.TCP connection establishment
b.TCP data transfer
c.TCP connection termination

TCP connection establishment
The following scenario occurs when a TCP connection is established:
1. The server must be prepared to accept an incoming connection. This is normally done by calling socket, bind, and listen and is called a passive open.

2. The client issues an active open by calling connect. This causes the client TCP to send a "synchronize" (SYN) segment, which tells the server the client's initial sequence number for the data that the client will send on the connection. Normally, there is no data sent with the SYN; it just contains an IP header, a TCP header, and possible TCP options (which we will talk about shortly).

3. The server must acknowledge (ACK) the client's SYN and the server must also send its own SYN containing the initial sequence number for the data that the server will send on the connection. The server sends its SYN and the ACK of the client's SYN in a single segment.

4.The client must acknowledge the server's SYN.
The minimum number of packets required for this exchange is three; hence, this is called TCP's three-way handshak*e*.



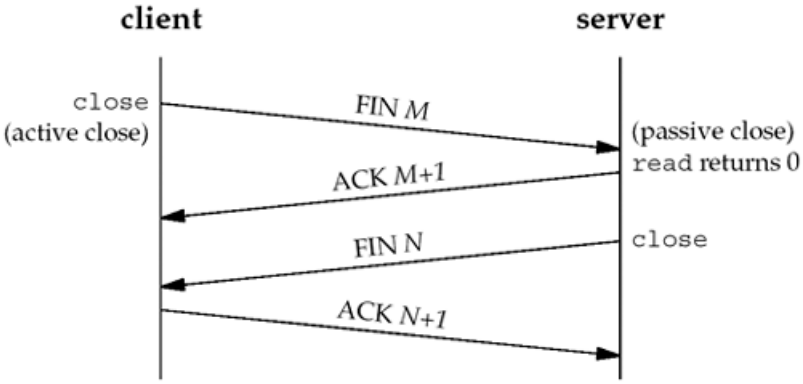
We show the client's initial sequence number as J and the server's initial sequence number as K. The acknowledgment number in an ACK is the next expected sequence number for the end sending the ACK. Since a SYN occupies one byte of the sequence number space, the acknowledgment number in the ACK of each SYN is the initial sequence number plus one. Similarly, the ACK of each FIN is the sequence number of the FIN plus one.

TCP connection termination
1. One application calls close first, and we say that this end performs the *active close*. This end's TCP sends a FIN segment, which means it is finished sending data.

2. The other end that receives the FIN performs the *passive close*. The received FIN is acknowledged by TCP. The receipt of the FIN is also passed to the application as an end-of-file , since the receipt of the FIN means the application will not receive any additional data on the connection.

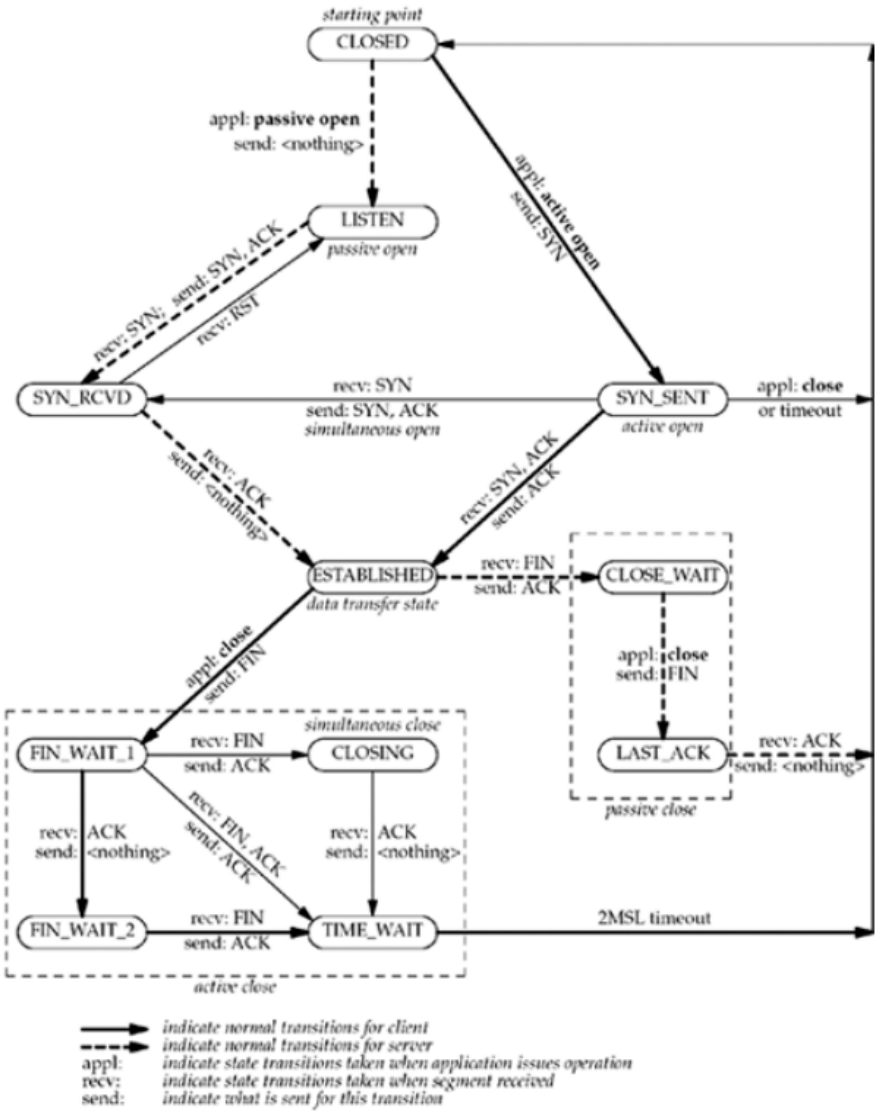
3. Sometime later, the application that received the end-of-file will close its socket. This causes its TCP to send a FIN.

4. The TCP on the system that receives this final FIN (the end that did the active close) acknowledges the FIN.



A FIN occupies one byte of sequence number space just like a SYN. Therefore, the ACK of each FIN is the sequence number of the FIN plus one.
Between Steps 2 and 3 it is possible for data to flow from the end doing the passive close to the end doing the active close. This is called a half-close.
The sending of each FIN occurs when a socket is closed

Q. Explain the TCP State Transition diagram with a neat diagram.



UNIT 2

With a standard POSIX definition explain socket address porotype for IPv4 *(sockaddr_in)* and IPv6 *(sockaddr_in6)*

POSIX is an acronym for Portable Operating System Interface. POSIX is not a single standard, but a
family of standards being developed by IEEE.

The POSIX specification requires only three members in the structure: sin_family, sin_addr, and sin_port. It is acceptable for a POSIX-compliant implementation to define additional structure members, and this is normal for an Internet socket address structure. Almost all implementations add the sin_zero member so that all socket address structures are at least 16 bytes in size.

IPV4(sockaddr_in)
An IPv4 socket address structure, commonly called an "Internet socket address structure," is named sockaddr_in and is defined by including the <netinet/in.h> header.

```
struct in_addr {
in_addr_t s_addr; /* 32-bit IPv4 address */
/* network byte ordered */
};
struct sockaddr_in {
uint8_t sin_len; /* length of structure (16) */
sa_family_t sin_family; /* AF_INET */
in_port_t sin_port; /* 16-bit TCP or UDP port number */
/* network byte ordered */
struct in_addr sin_addr; /* 32-bit IPv4 address */
/* network byte ordered */
char sin_zero[8]; /* unused */
};
```

IPV6(sockaddr_in6)

```
struct in6_addr {
uint8_t s6_addr[16]; /* 128-bit IPv6 address */
/* network byte ordered */
};
struct sockaddr_in6 {
uint8_t sin6_len; /* length of this struct (28) */
sa_family_t sin6_family; /* AF_INET6 */
in_port_t sin6_port; /* transport layer port# */
/* network byte ordered */
uint32_t sin6_flowinfo; /* flow information, undefined */
struct in6_addr sin6_addr; /* IPv6 address */
/* network byte ordered */
uint32_t sin6_scope_id; /* set of interfaces for a scope */
};
```

Q.With a standard POSIX definition explain socket address structure porotype: *sockaddr*

A socket address structures is always passed by reference when passed as an argument to any socket functions

A problem arises in how to declare the type of pointer that is passed. With ANSI C, the solution is simple: void * is the generic pointer type.

```
struct sockaddr {
uint8_t sa_len;
sa_family_t sa_family;          /* address family: AF_xxx value */
char sa_data[14];               /* protocol-specific address */
};
```

Q.Show the prototype for storage socket address structure: *sockaddr_storage*

- A new generic socket address structure was defined as part of the IPv6 sockets API, to overcome some of the shortcomings of the existing struct sockaddr.
- Unlike the struct sockaddr, the new struct sockaddr_storage is large enough to hold any socket address type supported by the system.

```
struct sockaddr_storage {
uint8_t ss_len;          /* length of this struct (implementation dependent) */
sa_family_t ss_family;          /* address family: AF_xxx value */
};
```

The sockaddr_storage type provides a generic socket address structure that is different from struct sockaddr in two ways:

- If any socket address structures that the system supports have alignment requirements, the sockaddr_storage provides the strictest alignment requirement.
- The sockaddr_storage is large enough to contain any socket address structure that the system supports.

Q.What are Value- Result Arguments? Explain the scenario with a neat block diagram.

When a socket address structure is passed to any socket function, it is always passed by reference. That is, a pointer to the structure is passed. The length of the structure is also passed as an argument. But the way in which the length is passed depends on which direction the structure is being passed: from the process to the kernel, or vice versa.

Three functions, bind, connect, and sendto, pass a socket address structure from the process to the kernel. One argument to these three functions is the pointer to the socket address structure and another argument is the integer size of the structure.

Since the kernel is passed both the pointer and the size of what the pointer points to, it knows exactly how much data to copy from the process into the kernel.

Four functions, accept, recvfrom, getsockname, and getpeername, pass a socket address structure from the kernel to the process, the reverse direction from the previous scenario. Two of the arguments to these four functions are the pointer to the socket address structure along with a pointer to an integer containing the size of the structure.
The reason that the size changes from an integer to be a pointer to an integer is because the size is both a value when the function is called (it tells the kernel the size of the structure so that the kernel does not write past the end of the structure when filling it in) and a result when the function returns (it tells the process how much information the kernel actually stored in the structure). This type of argument is called a value-result argument.

Figure 3.7. Socket address structure passed from process to kernel.

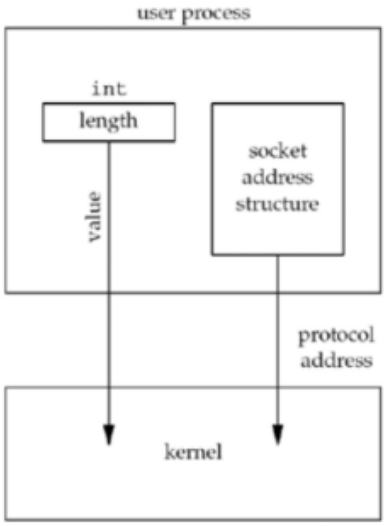
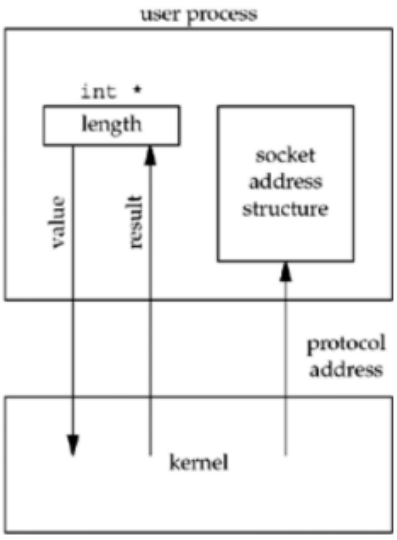


Figure 3.8. Socket address structure passed from kernel to process.



Q.Explain the functions which passes socket address structure from the process to the kernel with a neat block diagram.<DK>

- bind Function :

The bind function assigns a local protocol address to a socket.
With the Internet protocols, the protocol address is the combination of either a 32-bit IPv4 address or a 128-bit IPv6 address, along with a 16-bit TCP or UDP port number.

```
#include <sys/socket.h>
int bind (int sockfd, const struct sockaddr *myaddr, socklen_t addrlen);
```

Returns: 0 if OK,-1 on error
bind assigns a protocol address to a socket, and what that protocol address means depends on the protocol. The second argument is a pointer to a protocol-specific address, and the third argument is the size of this address structure. With TCP, calling bind lets us specify a port number, an IP address, both, or neither.

(Servers bind their well-known port when they start.
If a TCP client or server does not do this, the kernel chooses an ephemeral port for the socket when either connect or listen is called. It is normal for a TCP client to let the kernel choose an ephemeral port, unless the application requires a reserved port, but it is rare for a TCP server to let the kernel choose an ephemeral port, since servers are known by their well-known port.
A process can bind a specific IP address to its socket. The IP address must belong to an interface on the host. For a TCP client, this assigns the source IP address that will be used for IP datagrams sent on the socket. For a TCP server, this restricts the socket to receive incoming client connections destined only to that IP address. Normally, a TCP client does not bind an IP address to its socket. The kernel chooses the source IP address when the socket is connected, based on the outgoing interface that is used, which in turn is based on the route required to reach the server.)

Process specifies		Result
IP address	port	
Wildcard	0	Kernel chooses IP address and port
Wildcard	nonzero	Kernel chooses IP address, process specifies port
Local IP address	0	Process specifies IP address, kernel chooses port
Local IP address	nonzero	Process specifies IP address and port

If we specify a port number of 0, the kernel chooses an ephemeral port when bind is called. But if we specify a wildcard IP address, the kernel does not choose the local IP address until either the socket is connected (TCP) or a datagram is sent on the socket (UDP). With IPv4, the wildcard address is specified by the constant INADDR_ANY, whose value is normally 0. This tells the kernel to choose the IP address.

- connect function :

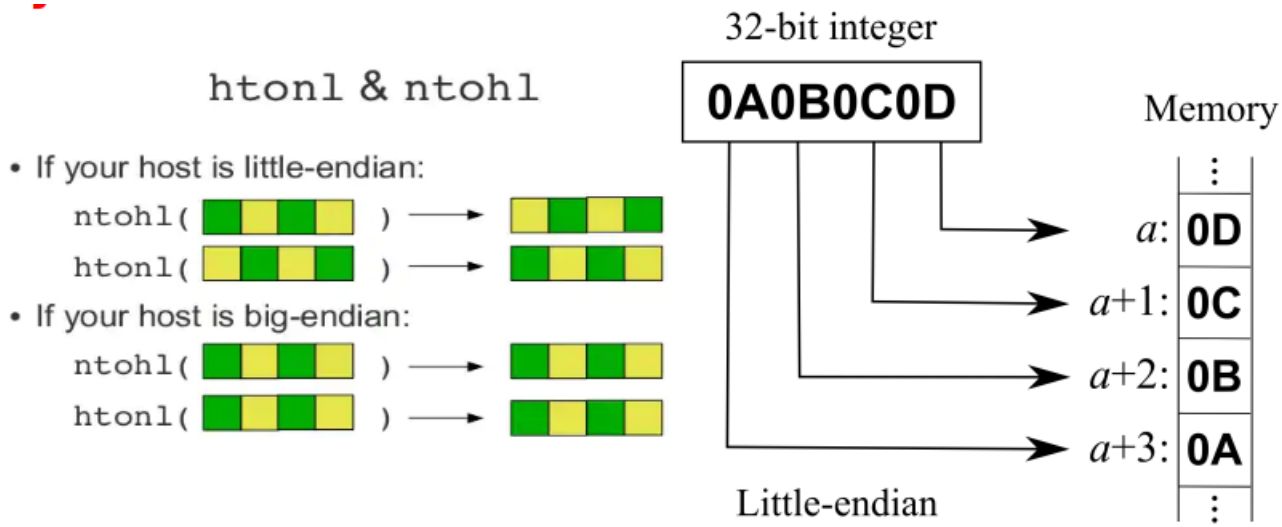
The connect function is used by a TCP client to establish a connection with a TCP server.

```
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);
```

Returns: 0 if OK, -1 on error

Q.Explain with a neat diagram the various byte ordering functions.<DK>

Functions are
htons(): host to network short — This function converts 16-bit quantities from host byte order to network byte order.
htonl(): host to network long — This function converts 32-bit quantities from host byte order to network byte order.
ntohs(): network to host short — This function converts 16-bit quantities from network byte order to host byte order.
ntohl(): network to host long — This function converts 32-bit quantities from network byte order to host byte order.



Q.Write a note on the Byte Manipulation Functions.

There are two groups of functions that operate on multibyte fields, without interpreting the data, and without assuming that the data is a null-terminated C string. We need these types of functions when dealing with socket address structures because we need to manipulate fields such as IP addresses, which can contain bytes of 0, but are not C character strings. The functions beginning with str (for string), defined by including the header, deal with null-terminated C character strings.
The first group of functions, whose names begin with b (for byte), are from 4.2BSD and are still provided by almost any system that supports the socket functions. The second group of functions, whose names begin with mem (for memory), are from the ANSI C standard and are provided with any system that supports an ANSI C library.
We first show the Berkeley-derived functions, although the only one we use in this text is bzero. (We use it because it has only two arguments and is easier to remember than the three-argument memset function, as explained on p. 8.) You may encounter the other two functions, bcopy and bcmp, in existing applications.

```
#include <strings.h>
void bzero(void *dest, size_t nbytes);
void bcopy(const void *src, void *dest, size_t nbytes);
int bcmp(const void *ptr1, const void *ptr2, size_t nbytes);
Returns: 0 if equal, nonzero if unequal
```

Q) C program to determine host byte order

Figure 3.10 Program to determine host byte order.

intro/byteorder.c

```
 1 #include     "unp.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     union {
 6         short   s;
 7         char    c[sizeof(short)];
 8     } un;

 9     un.s = 0x0102;
10     printf("%s: ", CPU_VENDOR_OS);
11     if (sizeof(short) == 2) {
12         if (un.c[0] == 1 && un.c[1] == 2)
13             printf("big-endian\n");
14         else if (un.c[0] == 2 && un.c[1] == 1)


15             printf("little-endian\n");
16         else
17             printf("unknown\n");
18     } else
19         printf("sizeof(short) = %d\n", sizeof(short));

20     exit(0);
21 }
```

10.Illustrate the significance of socket functions for elementary TCP client/server with a neat block diagram.

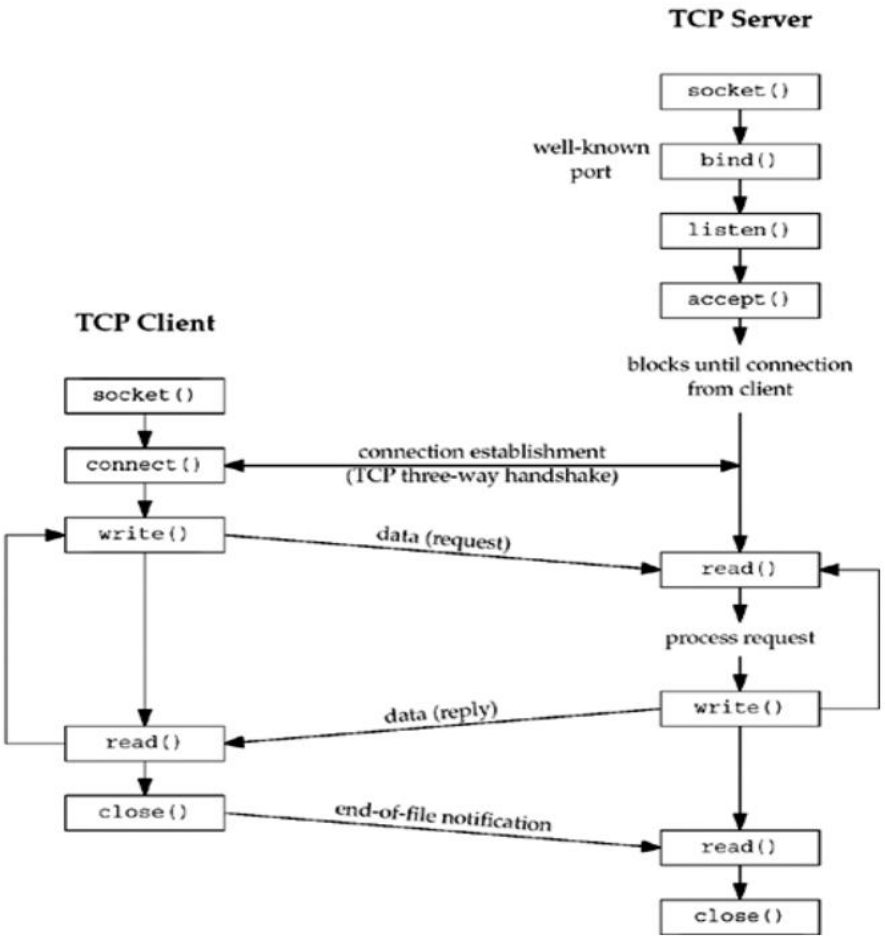


Figure 4.1 shows a timeline of the typical scenario that takes place between a TCP client and server. First, the server is started, then sometime later, a client is started that connects to the server. We assume that the client sends a request to the server, the server processes the request, and the server sends a reply back to the client. This continues until the client closes its end of the connection, which sends an end-of-file notification to the server. The server then closes its end of the connection and either terminates or waits for a new client connection.

11. Explain the following arguments of the socket function:

a. **Family**
b. **Type**
c. **Protocol**

4.2 socket Function

To perform network I/O, the first thing a process must do is call the socket function, specifying the type of communication protocol desired (TCP using IPv4, UDP using IPv6, Unix domain stream protocol, etc.).

```
#include <sys/socket.h>

int socket (int family, int type, int protocol);
```

Returns: non-negative descriptor if OK, -1 on error

family specifies the protocol family,

Protocol *family* constants for `socket` function.

family	Description
AF_INET	IPv4 protocols
AF_INET6	IPv6 protocols
AF_LOCAL	Unix domain protocols (Chapter 15)
AF_ROUTE	Routing sockets (Chapter 18)
AF_KEY	Key socket (Chapter 19)

The socket **type** is one of the constants.

type	Description
SOCK_STREAM	stream socket
SOCK_DGRAM	datagram socket
SOCK_SEQPACKET	sequenced packet socket
SOCK_RAW	raw socket

***protocol* of sockets for `AF_INET` or `AF_INET6`.**

Protocol	Description
IPPROTO_TCP	TCP transport protocol
IPPROTO_UDP	UDP transport protocol
IPPROTO_SCTP	SCTP transport protocol

12. Explain the following functions of TCP socket:

a. **connect**
b. **bind**
c. **listen**
d. **accept**
e. **close**

4.3 `connect` Function

The `connect` function is used by a TCP client to establish a connection with a TCP server.

```
#include <sys/socket.h>

int connect(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);
```

Returns: 0 if OK, -1 on error

sockfd is a socket descriptor returned by the `socket` function.

The second and third arguments are a pointer to a socket address structure and its size.

4.4 `bind` Function

The `bind` function assigns a local protocol address to a socket. With the Internet protocols, the protocol address is the combination of either a 32-bit IPv4 address or a 128-bit IPv6 address, along with a 16-bit TCP or UDP port number.

```
#include <sys/socket.h>

int bind (int sockfd, const struct sockaddr *myaddr, socklen_t addrlen);
```

Returns: 0 if OK,-1 on error

4.5 `listen` Function

The `listen` function is called only by a TCP server and it performs two actions:

1. When a socket is created by the `socket` function, it is assumed to be an active socket, that is, a client socket that will issue a `connect`. The `listen` function converts an unconnected socket into a passive socket, indicating that the kernel should accept incoming connection requests directed to this socket. In terms of the TCP state transition diagram (Figure 2.4), the call to `listen` moves the socket from the CLOSED state to the LISTEN state.
2. The second argument to this function specifies the maximum number of connections the kernel should queue for this socket.

```
#include <sys/socket.h>

#int listen (int sockfd, int backlog);
```

Returns: 0 if OK, -1 on error

This function is normally called after both the `socket` and `bind` functions and must be called before calling the `accept` function.

4.6 `accept` Function

`accept` is called by a TCP server to return the next completed connection from the front of the completed connection queue (Figure 4.7). If the completed connection queue is empty, the process is put to sleep (assuming the default of a blocking socket).

```
#include <sys/socket.h>

int accept (int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);
```

Returns: non-negative descriptor if OK, -1 on error

The cliaddr and addrlen arguments are used to return the protocol address of the connected peer process (the client).

4.9 close Function

The normal Unix close function is also used to close a socket and terminate a TCP connection.

```
#include <unistd.h>

int close (int sockfd);
```

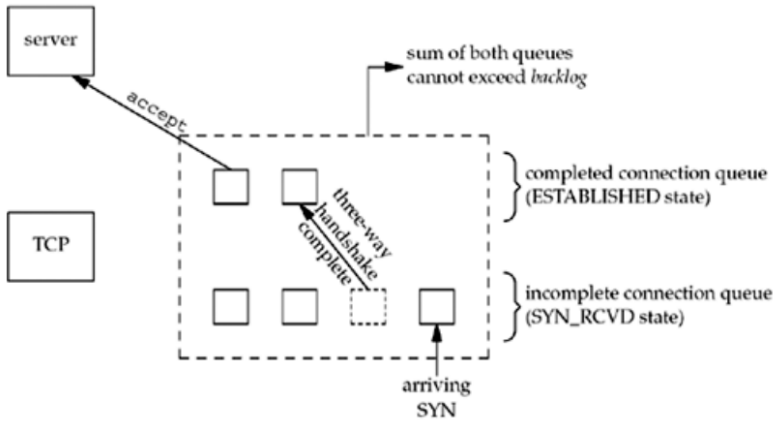
Returns: 0 if OK, -1 on error

13. With a neat block diagram explain the queues maintained by TCP for a listening socket. Also show the packets exchanged during the connection establishment with these two queues.

To understand the *backlog* argument, we must realize that for a given listening socket, the kernel maintains two queues:

1. An *incomplete connection queue*, which contains an entry for each SYN that has arrived from a client for which the server is awaiting completion of the TCP three-way handshake. These sockets are in the SYN_RCVD state (Figure 2.4).
2. A *completed connection queue*, which contains an entry for each client with whom the TCP three-way handshake has completed. These sockets are in the ESTABLISHED state (Figure 2.4).

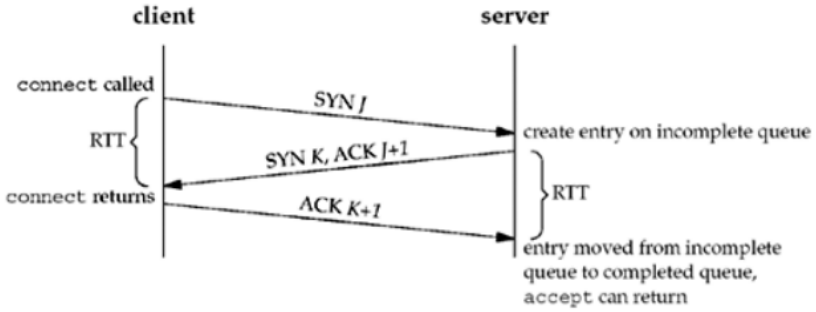
Figure 4.7. The two queues maintained by TCP for a listening socket.



When an entry is created on the incomplete queue, the parameters from the listen socket are copied over to the newly created connection. The connection creation mechanism is completely automatic; the server process is not involved.

packets exchanged during the connection establishment with these two queues.

Figure 4.8. TCP three-way handshake and the two queues for a listening socket.



- When a SYN arrives from a client, TCP creates a new entry on the incomplete queue and then responds with the second segment of the three-way handshake: the server's SYN with an ACK of the client's SYN
- This entry will remain on the incomplete queue until the third segment of the three-way handshake arrives (the client's ACK of the server's SYN), or until the entry times out.
- If the three-way handshake completes normally, the entry moves from the incomplete queue to the end of the completed queue.
- When the process calls accept, which we will describe in the next section, the first entry on the completed queue is returned to the process, or if the queue is empty, the process is put to sleep until an entry is placed onto the completed queue.

14.Illustrate the significance of fork and exec functions.

`fork` and `exec` Functions

```
#include <unistd.h>
```

```
pid_t fork(void);
```

Returns: 0 in child, process ID of child in parent, -1 on error

- It returns once in the calling process (called the parent) with a return value that is the process ID of the newly created process (the child). It also returns once in the child, with a return value of 0. Hence, the return value tells the process whether it is the parent or the child.
- The reason **fork** returns 0 in the child, instead of the parent's process ID, is because a child has only one parent and it can always obtain the parent's process ID by calling **getppid**.

There are two typical uses of `fork`:

1. A process makes a copy of itself so that one copy can handle one operation while the other copy does another task. This is typical for network servers. We will see many examples of this later in the text.
2. A process wants to execute another program. Since the only way to create a new process is by calling `fork`, the process first calls `fork` to make a copy of itself, and then one of the copies (typically the child process) calls `exec` (described next) to replace itself with the new program. This is typical for programs such as shells.

exec function:

- The only way in which an executable program file on disk can be executed by Unix is for an existing process to call one of the six exec functions.
- **exec** replaces the current process image with the new program file, and this new program normally starts at the **main** function.

```
#include <unistd.h>
int execl (const char *pathname, const char *arg0, ... /* (char *) 0 */ );
int execv (const char *pathname, char *const argv[]);
int execle (const char *pathname, const char *arg0, ...
                    /* (char *) 0, char *const envp[] */ );
int execve (const char *pathname, char *const argv[], char *const envp[]);
int execlp (const char *filename, const char *arg0, ... /* (char *) 0 */ );
int execvp (const char *filename, char *const argv[]);
                                        All six return: -1 on error, no return on success
```

15.Outline the typical concurrent server with the help of pseudocode.

The simplest way to write a concurrent server under Unix is to fork a child process to handle each client.

Figure 4.13 Outline for typical concurrent server.

```
pid_t pid;
int   listenfd,  connfd;

listenfd = Socket( ... );

    /* fill in sockaddr_in{} with server's well-known port */
Bind(listenfd, ... );
Listen(listenfd, LISTENQ);

for ( ; ; ) {
    connfd = Accept (listenfd, ... );    /* probably blocks */

    if( (pid = Fork()) == 0) {
       Close(listenfd);    /* child closes listening socket */
       doit(connfd);       /* process the request */
       Close(connfd);      /* done with this client */
       exit(0);            /* child terminates */
    }

    Close(connfd);         /* parent closes connected socket */
}
```

When a connection is established, `accept` returns, the server calls `fork`, and the child process services the client (on `connfd`, the connected socket) and the parent process waits for another connection (on `listenfd`, the listening socket). The parent closes the connected socket since the child handles the new client.

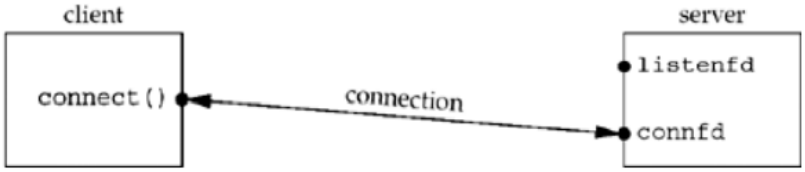
16.Demonstrate the status of client/ server before and after call to *accept* returns with a neat block diagram.

Figure 4.14. Status of client/server before call to `accept` returns.

Above figure shows the status of the client and server while the server is blocked in the call to accept and the connection request arrives from the client.

Immediately after accept returns, we have the scenario shown in Figure 4.15. The connection is accepted by the kernel and a new socket, connfd, is created. This is a connected socket and data can now be read and written across the connection.

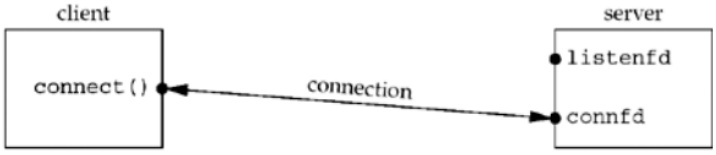
Figure 4.15. Status of client/server after return from `accept`.



17.Demonstrate the status of client/ server after fork returns with a neat block diagram.

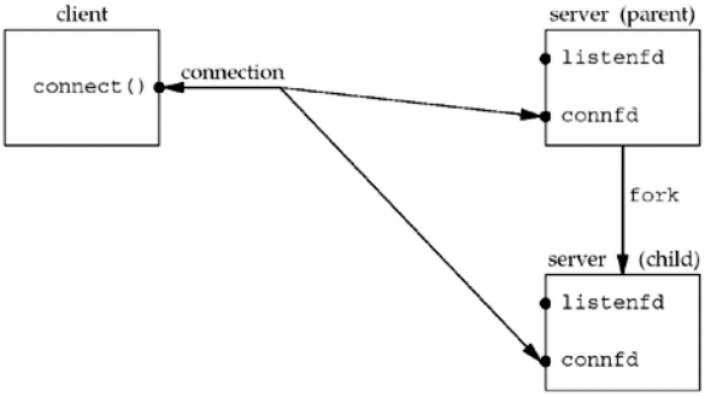
Immediately after `accept` returns, we have the scenario shown in Figure 4.15. The connection is accepted by the kernel and a new socket, `connfd`, is created. This is a connected socket and data can now be read and written across the connection.

Figure 4.15. Status of client/server after return from `accept`.



The next step in the concurrent server is to call `fork`. Figure 4.16 shows the status after `fork` returns.

Figure 4.16. Status of client/server after `fork` returns.

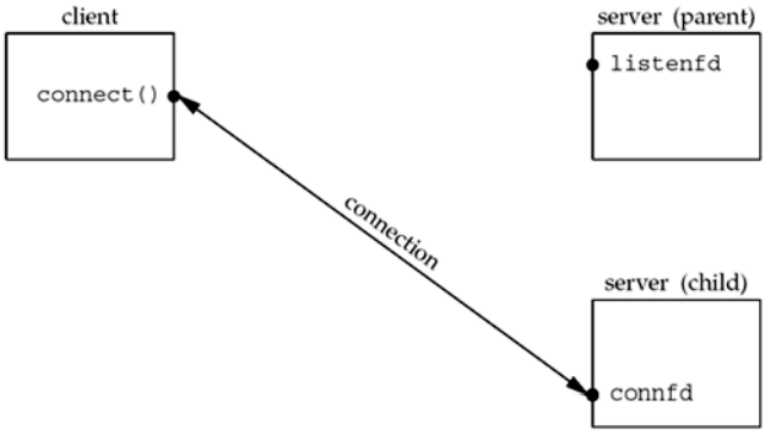


Notice that both descriptors, `listenfd` and `connfd`, are shared (duplicated) between the parent and child.

18.Demonstrate the status of client/ server after parent and child close appropriate sockets with a neat block diagram.

The next step is for the parent to close the connected socket and the child to close the listening socket. This is shown in Figure 4.17.

Figure 4.17. Status of client/server after parent and child close appropriate sockets.



This is the desired final state of the sockets. The child is handling the connection with the client and the parent can call `accept` again on the listening socket, to handle the next client connection.

19.Comment on the significance of *getsockname* and *getpeername* functions.

4.10 `getsockname` and `getpeername` Functions

These two functions return either the local protocol address associated with a socket (`getsockname`) or the foreign protocol address associated with a socket (`getpeername`).

`#include <sys/socket.h>`
`int getsockname(int sockfd, struct sockaddr *localaddr, socklen_t *addrlen);`
`int getpeername(int sockfd, struct sockaddr *peeraddr, socklen_t *addrlen);`
Both return: 0 if OK, -1 on error

Notice that the final argument for both functions is a value-result argument. That is, both functions fill in the socket address structure pointed to by *localaddr* or *peeraddr*.

These two functions are required for the following reasons:

- After `connect` successfully returns in a TCP client that does not call `bind`, `getsockname` returns the local IP address and local port number assigned to the connection by the kernel.
- After calling `bind` with a port number of 0 (telling the kernel to choose the local port number), `getsockname` returns the local port number that was assigned.
- `getsockname` can be called to obtain the address family of a socket, as we show in
- When a server is `exec`ed by the process that calls `accept`, the only way the server can obtain the identity of the client is to call `getpeername`. This is what happens

20.Develop the pseudocode that returns the address family of a socket.

The `sockfd_to_family` function shown in Figure 4.19 returns the address family of a socket.

Figure 4.19 Return the address family of a socket.

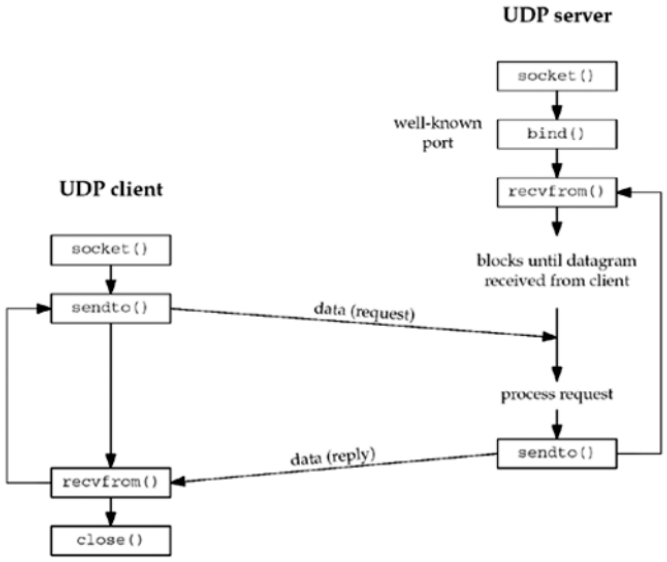
lib/sockfd_to_family.c

```
 1 #include    "unp.h"
 2 int
 3 sockfd_to_family(int sockfd)
 4 {
 5     struct sockaddr_storage ss;
 6     socklen_t len;

 7     len = sizeof(ss);
 8     if (getsockname(sockfd, (SA *) &ss, &len) < 0)
 9         return (-1);
10     return (ss.ss_family);
11 }
```

UNIT – 3

1. Illustrate the significance of socket functions for UDP TCP client/server with a neat block diagram.

Figure 8.1. Socket functions for UDP client/server.



- The Figure shows the function calls for a typical UDP client/server.
- The client does not establish a connection with the server. Instead, the client just sends a datagram to the server using the **sendto** function (described in the next section), which requires the address of the destination (the server) as a parameter.
- Similarly, the server does not accept a connection from a client. Instead, the server just calls the **recvfrom** function, which waits until data arrives from some client.
- **recvfrom** returns the protocol address of the client, along with the datagram, so the server can send a response to the correct client.
- Figure shows a timeline of the typical scenario that takes place for a UDP client/server exchange.

2.Explain the following functions of UDP socket:

- **recvfrom**
- **sendto**

These two functions are similar to the standard `read` and `write` functions, but three additional arguments are required.

```
#include <sys/socket.h>
```

```
ssize_t recvfrom(int sockfd, void *buff, size_t nbytes, int flags, struct sockaddr *
from, socklen_t *addrlen);
```

```
ssize_t sendto(int sockfd, const void *buff, size_t nbytes, int flags, const struct
sockaddr *to, socklen_t addrlen);
```

Both return: number of bytes read or written if OK, 1 on error

- The first three arguments, **sockfd**, **buff**, and **nbytes**, are identical to the first three arguments for **read** and **write**: descriptor, pointer to buffer to read into or write from, and number of bytes to read or write.
- Both functions return the length of the data that was read or written as the value of the function. In the typical use of **recvfrom**, with a datagram protocol, the return value is the amount of user data in the datagram received.

3. List and explain with a neat block diagram the steps associated with simple UDP echo client and server.

4. Develop the 'C' program to demonstrate the UDP echo server: main function

Figure 8.2. Simple echo client/server using UDP.

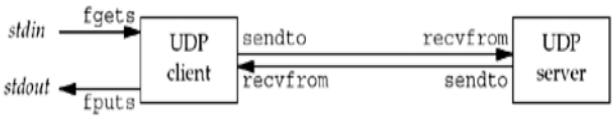


Figure 8.3 UDP echo server.

udpcliserv/udpserv01.c

```
 1 #include     "unp.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     sockfd;
 6     struct sockaddr_in servaddr, cliaddr;

 7     sockfd = Socket(AF_INET, SOCK_DGRAM, 0);

 8     bzero(&servaddr, sizeof(servaddr));
 9     servaddr.sin_family = AF_INET;
10     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
11     servaddr.sin_port = htons(SERV_PORT);

12     Bind(sockfd, (SA *) &servaddr, sizeof(servaddr));

13     dg_echo(sockfd, (SA *) &cliaddr, sizeof(cliaddr));
14 }
```

Create UDP socket, bind server's well-known port

7 12 We create a UDP socket by specifying the second argument to `socket` as `SOCK_DGRAM` (a datagram socket in the IPv4 protocol). As with the TCP server example, the IPv4 address for the `bind` is specified as `INADDR_ANY` and the server's well-known port is the constant `SERV_PORT` from the `unp.h` header.

13 The function `dg_echo` is called to perform server processing.

5. Develop the 'C' program to demonstrate the UDP echo server: dg_echo function

Figure 8.4 dg_echo function: echo lines on a datagram socket.

lib/dg_echo.c

```
 1 #include     "unp.h"

 2 void
 3 dg_echo(int sockfd, SA *pcliaddr, socklen_t clilen)
 4 {
 5     int     n;
 6     socklen_t len;
 7     char    mesg[MAXLINE];

 8     for ( ; ; ) {
 9         len = clilen;
10         n = Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);

11         Sendto(sockfd, mesg, n, 0, pcliaddr, len);
12     }
13 }
```

8 12 This function is a simple loop that reads the next datagram arriving at the server's port using `recvfrom` and sends it back using `sendto`.

Next, this function provides an *iterative server*, not a concurrent server as we had with TCP. There is no call to `fork`, so a single server process handles any and all clients. In general, most TCP servers are concurrent and most UDP servers are iterative.

6. Develop the 'C' program to demonstrate the UDP echo client: main function

Figure 8.7 UDP echo client.

udpcliserv/udpcli01.c

```
 1 #include     "unp.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     sockfd;
 6     struct sockaddr_in servaddr;

 7     if(argc != 2)
 8        err_quit("usage: udpcli <IPaddress>");

 9     bzero(&servaddr, sizeof(servaddr));
10     servaddr.sin_family = AF_INET;
11     servaddr.sin_port = htons(SERV_PORT);
12     Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

13     sockfd = Socket(AF_INET, SOCK_DGRAM, 0);

14     dg_cli(stdin, sockfd, (SA *) &servaddr, sizeof(servaddr));

15     exit(0);
16 }
```

9 12 An IPv4 socket address structure is filled in with the IP address and port number of the server. This structure will be passed to `dg_cli`, specifying where to send datagrams.

13 14 A UDP socket is created and the function `dg_cli` is called.

7. Develop the 'C' program to demonstrate the UDP echo client: dg_cli function

Figure 8.8 `dg_cli` function: client processing loop.

lib/dg_cli.c

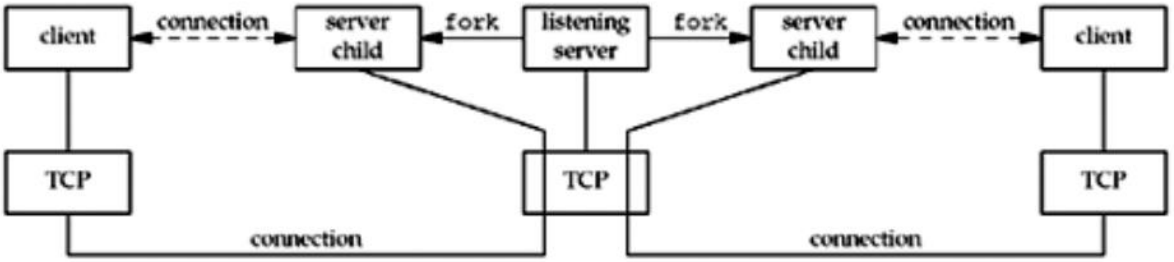
```
 1 #include     "unp.h"

 2 void
 3 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
 4 {
 5     int     n;
 6     char    sendline[MAXLINE], recvline[MAXLINE + 1];

 7     while (Fgets(sendline, MAXLINE, fp) != NULL) {

 8         Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);

 9         n = Recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);

10         recvline[n] = 0;        /* null terminate */
11         Fputs(recvline, stdout);
12     }
13 }
```

7 12 There are four steps in the client processing loop: read a line from standard input using `fgets`, send the line to the server using `sendto`, read back the server's echo using `recvfrom`, and print the echoed line to standard output using `fputs`.

Notice that the call to recvfrom specifies a null pointer as the fifth and sixth arguments. This tells the kernel that we are not interested in knowing who sent the reply.

8. Outline the summary of TCP client/server with two clients.

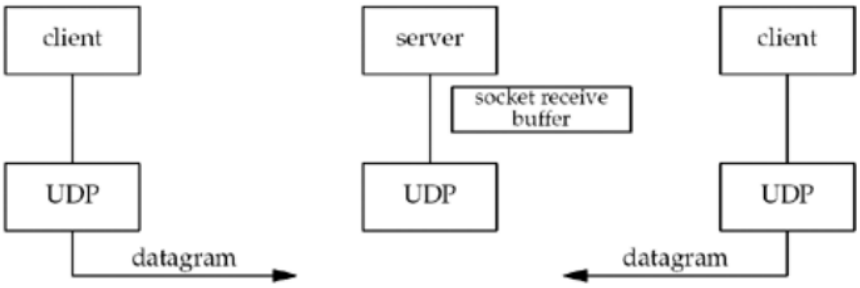
Figure 8.5. Summary of TCP client/server with two clients.



There are two connected sockets and each of the two connected sockets on the server host has its own socket receive buffer.

9. Outline the summary of UDP client/server with two clients.

Figure 8.6. Summary of UDP client/server with two clients.



There is only one server process and it has a single socket on which it receives all arriving datagrams and sends all responses. That socket has a receive buffer into which all arriving datagrams are placed.

10. Develop the 'C' program for dg_cli function that verifies returned socket address.

Figure 8.8 `dg_cli` function: client processing loop.

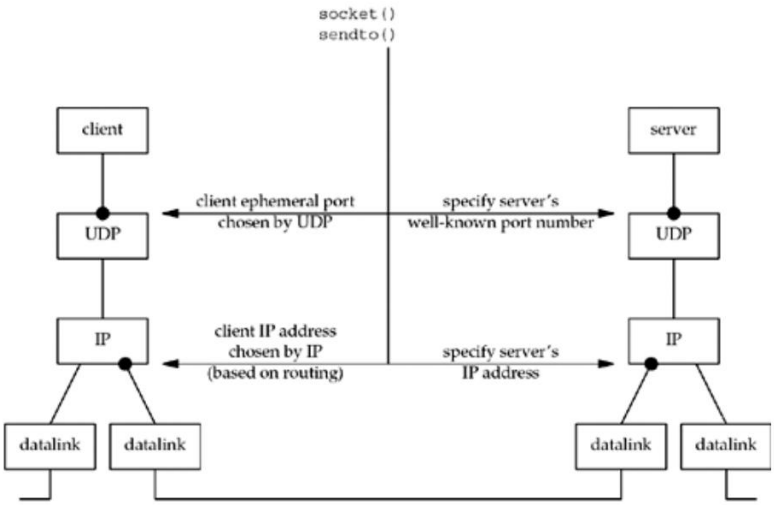
lib/dg_cli.c

```
 1 #include     "unp.h"

 2 void
 3 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
 4 {
 5     int     n;
 6     char    sendline[MAXLINE], recvline[MAXLINE + 1];

 7     while (Fgets(sendline, MAXLINE, fp) != NULL) {

 8         Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);

 9         n = Recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);

10         recvline[n] = 0;        /* null terminate */
11         Fputs(recvline, stdout);
12     }
13 }
```

7 12 There are four steps in the client processing loop: read a line from standard input using `fgets`, send the line to the server using `sendto`, read back the server's echo using `recvfrom`, and print the echoed line to standard output using `fputs`.

11. Outline the summary of UDP client/server from client's perspective with a neat block diagram.

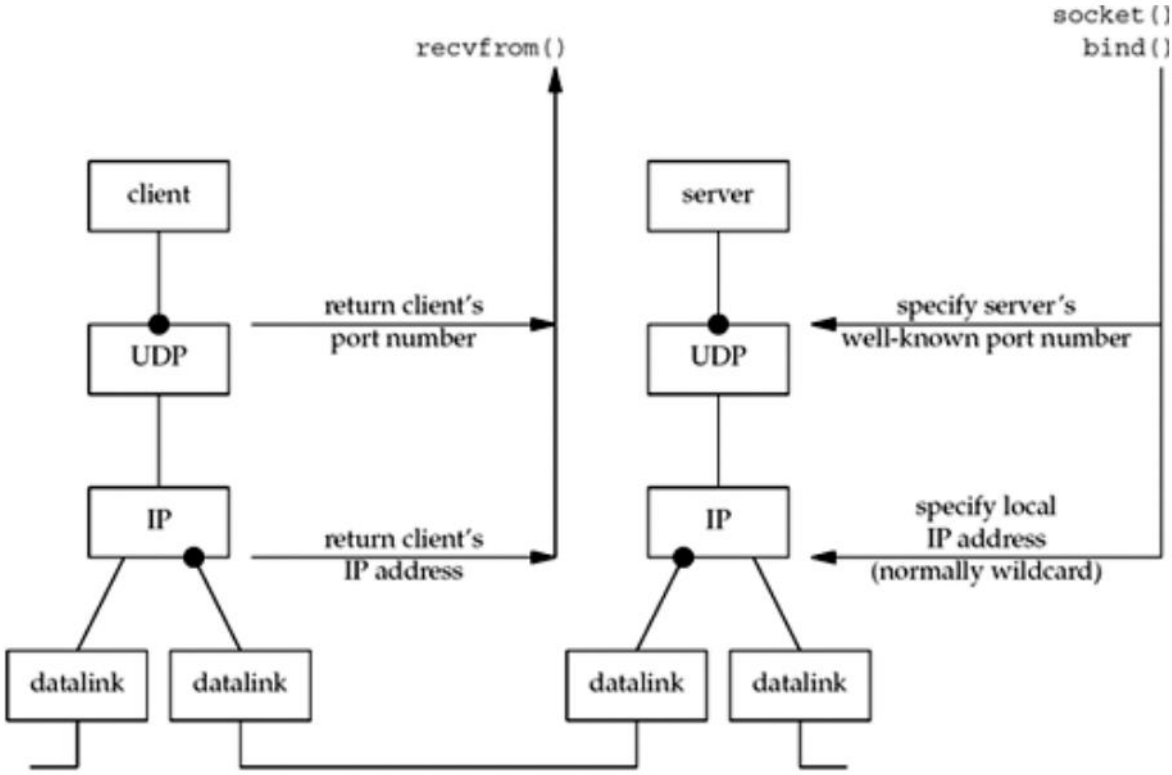
Figure 8.11. Summary of UDP client/server from client's perspective.



- The client must specify the server's IP address and port number for the call to **sendto**. Normally, the client's IP address and port are chosen automatically by the kernel, although we mentioned that the client can call **bind** if it so chooses.
- If these two values for the client are chosen by the kernel, we also mentioned that the client's ephemeral port is chosen once, on the first **sendto**, and then it never changes.
- The client's IP address, however, can change for every UDP datagram that the client sends, assuming the client does not bind a specific IP address to the socket.
- The reason is shown in Figure : If the client host is multihomed, the client could alternate between two destinations, one going out the datalink on the left, and the other going out the datalink on the right.
- What happens if the client **binds** an IP address to its socket, but the kernel decides that an outgoing datagram must be sent out some other datalink? In this case the IP datagram will contain a source IP address that is different from the IP address of the outgoing datalink

12. Outline the summary of UDP client/server from server's perspective with a neat block diagram.

Figure 8.12. Summary of UDP client/server from server's perspective.



There are at least four pieces of information that a server might want to know from an arriving IP datagram: **the source IP address**, **destination IP address**, **source port number**, and **destination port number.** Figure 8.13 shows the function calls that return this information for a TCP server and a UDP server.

Figure 8.13. Information available to server from arriving IP datagram.

From client's IP datagram	TCP server	UDP server
Source IP address	`accept`	`recvfrom`
Source port number	`accept`	`recvfrom`
Destination IP address	`getsockname`	`recvmsg`
Destination port number	`getsockname`	`getsockname`

13. Develop the 'C' program to demonstrate the UDP dg_cli function that calls connect.

Figure 8.17 dg_cli function that calls connect.

udpcliserv/dgcliconnect.c

```
 1 #include     "unp.h"

 2 void
 3 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
 4 {
 5     int     n;
 6     char    sendline[MAXLINE], recvline[MAXLINE + 1];

 7     Connect(sockfd, (SA *) pservaddr, servlen);

 8     while (Fgets(sendline, MAXLINE, fp) != NULL) {

 9         Write(sockfd, sendline, strlen(sendline));

10         n = Read(sockfd, recvline, MAXLINE);

11         recvline[n] = 0;        /* null terminate */
12         Fputs(recvline, stdout);
13     }
14 }
```

14. Develop the 'C' program to demonstrate the UDP dg_cli function that writes a fixed number of datagrams to the server.

Figure 8.19 dg_cli function that writes a fixed number of datagrams to the server.

udpcliserv/dgcliloop1.c

```
 1 #include     "unp.h"

 2 #define NDG      2000          /*  datagrams to send */
 3 #define DGLEN    1400          /*  length of each datagram */

 4 void
 5 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
 6 {
 7     int     i;
 8     char    sendline[DGLEN];

 9     for (i = 0; i < NDG; i++) {
10         Sendto(sockfd, sendline, DGLEN, 0, pservaddr, servlen);
11     }
12 }
```

15. Develop the 'C' program to demonstrate the UDP dg_echo function that counts received datagrams.

Figure 8.20 dg_echo function that counts received datagrams.

udpcliserv/dgecholoop1.c

```
 1 #include     "unp.h"

 2 static void recvfrom_int(int);
 3 static int count;

 4 void
 5 dg_echo(int sockfd, SA *pcliaddr, socklen_t clilen)
 6 {
 7     socklen_t len;
 8     char    mesg[MAXLINE];

 9     Signal(SIGINT, recvfrom_int);

10     for ( ; ; ) {
11         len = clilen;
12         Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);

13         count++;
14     }
15 }

16 static void
17 recvfrom_int(int signo)
18 {
19     printf("\nreceived %d datagrams\n", count);
20     exit(0);
21 }
```

16. Develop the 'C' program to demonstrate the UDP dg_echo function that increases the size of the socket receive queue.

Figure 8.22 dg_echo function that increases the size of the socket receive queue.

udpcliserv/dgecholoop2.c

```
 1 #include    "unp.h"

 2 static void recvfrom_int(int);
 3 static int count;

 4 void
 5 dg_echo(int sockfd, SA *pcliaddr, socklen_t clilen)
 6 {
 7     int     n;
 8     socklen_t len;
 9     char    mesg[MAXLINE];

10     Signal(SIGINT, recvfrom_int);

11     n = 220 * 1024;
12     Setsockopt(sockfd, SOL_SOCKET, SO_RCVBUF, &n, sizeof(n));

13     for ( ; ; ) {
14         len = clilen;
15         Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);

16         count++;
17     }
18 }

19 static void
20 recvfrom_int(int signo)
21 {
22     printf("\nreceived %d datagrams\n", count);


23     exit(0);
24 }
```

17. Develop the 'C' program for UDP that uses connect to determine outgoing interface.

Figure 8.23 UDP program that uses `connect` to determine outgoing interface.

udpcliserv/udpcli09.c

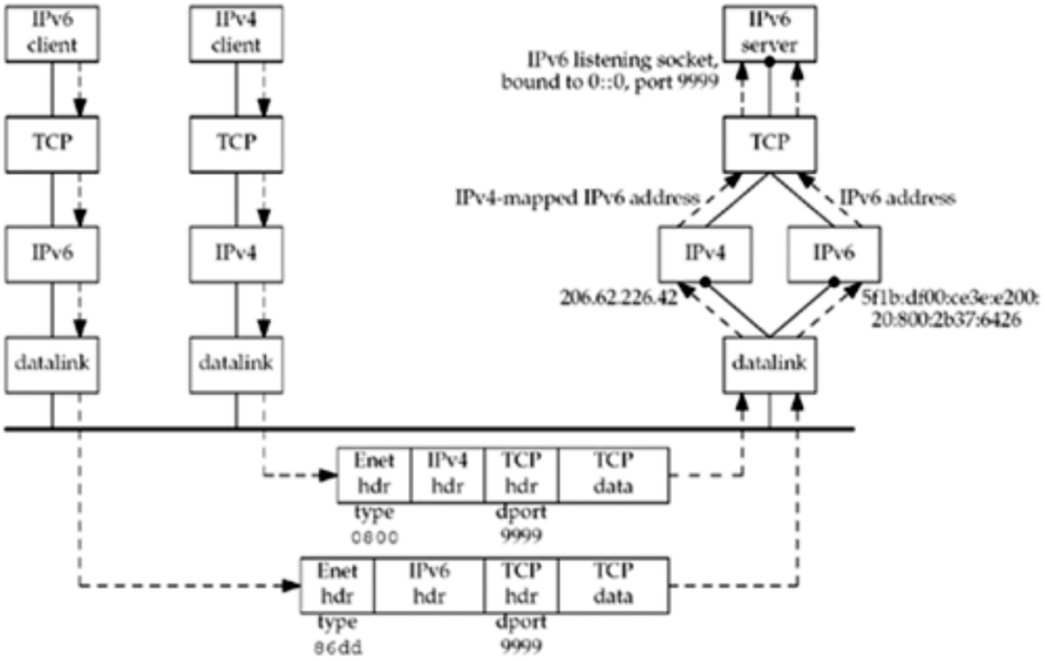
```
 1 #include     "unp.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     sockfd;
 6     socklen_t len;
 7     struct sockaddr_in cliaddr, servaddr;

 8     if (argc != 2)
 9         err_quit("usage: udpcli <IPaddress>");

10     sockfd = Socket(AF_INET, SOCK_DGRAM, 0);

11     bzero(&servaddr, sizeof(servaddr));
12     servaddr.sin_family = AF_INET;
13     servaddr.sin_port = htons(SERV_PORT);
14     Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

15     Connect(sockfd, (SA *) &servaddr, sizeof(servaddr));

16     len = sizeof(cliaddr);
17     Getsockname(sockfd, (SA *) &cliaddr, &len);
18     printf("local address %s\n", Sock_ntop((SA *) &cliaddr, len));

19     exit(0);
20 }
```

18. Make use of select function for TCP and UDP Echo server.

1. **With a neat block diagram explain IPv6 server on dual stack host serving IPv4 and IPv6 clients.**

A general property of a dual-stack host is that IPv6 servers can handle both IPv4 and IPv6 clients. This is done using IPv4-mapped IPv6 addresses. Figure 12.2 shows an example of this.

Figure 12.2. IPv6 server on dual-stack host serving IPv4 and IPv6 clients.



We have an IPv4 client and an IPv6 client on the left. The server on the right is written using IPv6 and it is running on a dual-stack host. The server has created an IPv6 listening TCP socket that is bound to the IPv6 wildcard address and TCP port 9999.

We can summarize the steps that allow an IPv4 TCP client to communicate with an IPv6 server as follows:

i. The IPv6 server starts, creates an IPv6 listening socket, and we assume it binds the wildcard address to the socket.

ii. The IPv4 client calls gethostbyname and finds an A record for the server. The server host will have both an A record and a AAAA record since it supports both protocols, but the IPv4 client asks for only an A record.

iii. The client calls connect and the client's host sends an IPv4 SYN to the server.

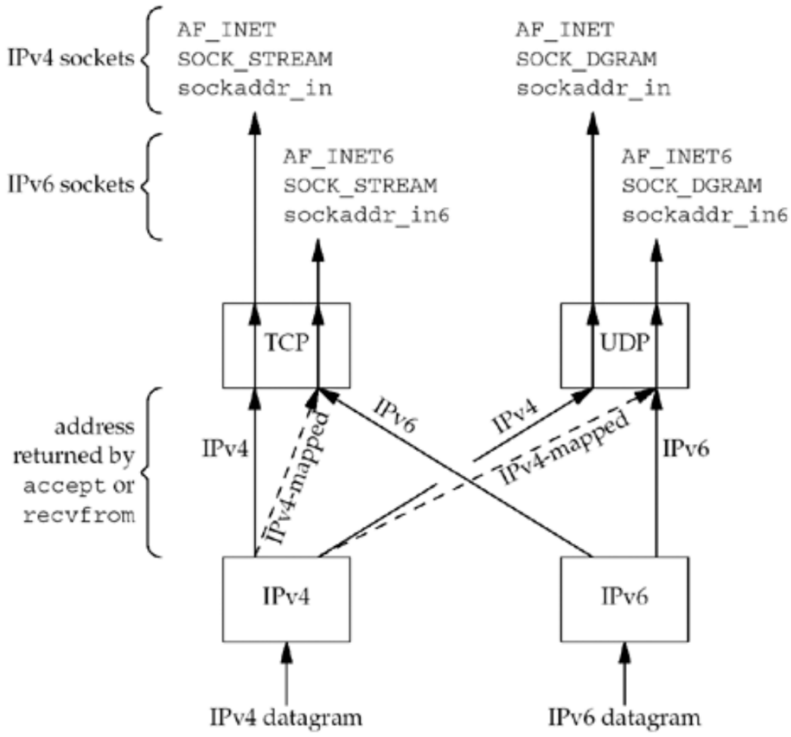
iv. The server host receives the IPv4 SYN directed to the IPv6 listening socket, sets a flag indicating that this connection is using IPv4-mapped IPv6 addresses, and responds with an IPv4 SYN/ACK. When the connection is established, the address returned to the server by *accept* is the IPv4-mapped IPv6 address.

v. When the server host sends to the IPv4-mapped IPv6 address, its IP stack generates IPv4 datagrams to the IPv4 address. Therefore, all communication between this client and server takes place using IPv4 datagrams.

vi. Unless the server explicitly checks whether this IPv6 address is an IPv4-mapped IPv6 address the server never knows that it is communicating with an IPv4 client. The dual-protocol stack handles this detail. Similarly, the IPv4 client has no idea that it is communicating with an IPv6 server.

2. Explain with a neat block diagram how the received IPv4 and IPv6 datagrams are processed depending on the type of receiving socket.

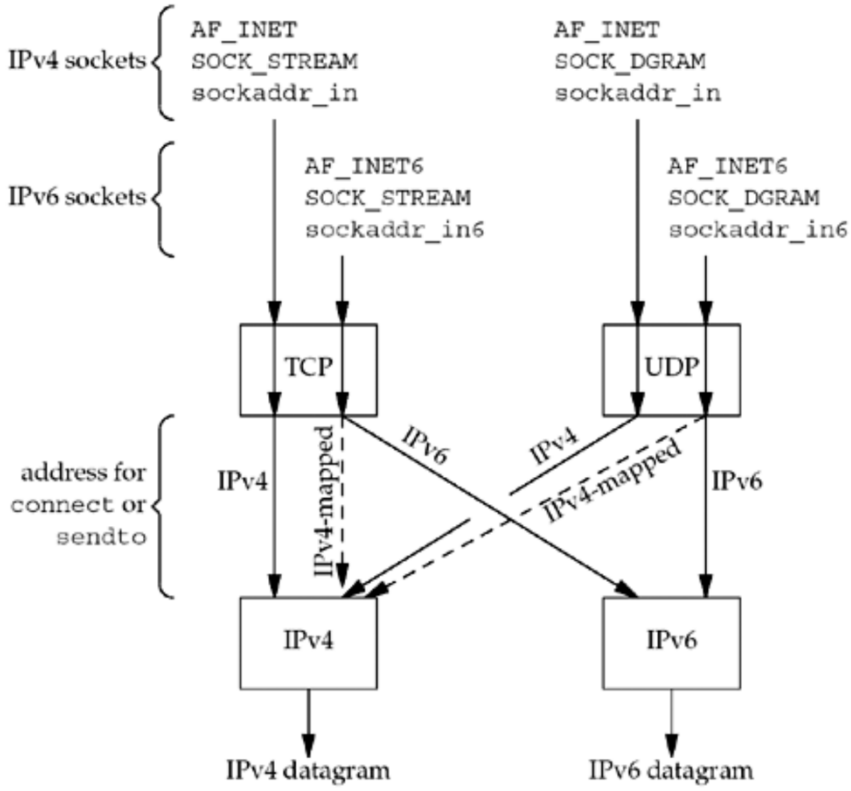
Figure 12.3. Processing of received IPv4 or IPv6 datagrams, depending on type of receiving socket.



- If an IPv4 datagram is received for an IPv4 socket, nothing special is done. These are the two arrows labeled "IPv4" in the figure: one to TCP and one to UDP. IPv4 datagrams are exchanged between the client and server.
- If an IPv6 datagram is received for an IPv6 socket, nothing special is done. These are the two arrows labeled "IPv6" in the figure: one to TCP and one to UDP. IPv6 datagrams are exchanged between the client and server.
- When an IPv4 datagram is received for an IPv6 socket, the kernel returns the corresponding IPv4-mapped IPv6 address as the address returned by accept (TCP) or recvfrom (UDP). These are the two dashed arrows in the figure. This mapping is possible because an IPv4 address can always be represented as an IPv6 address. IPv4 datagrams are exchanged between the client and server.
- The converse of the previous bullet is false: In general, an IPv6 address cannot be represented as an IPv4 address; therefore, there are no arrows from the IPv6 protocol box to the two IPv4 sockets

3. **Explain with a neat block diagram how the client requests are processed depending on the address type and socket type.**

Figure 12.4. Processing of client requests, depending on address type and socket type.



- An IPv4 server starts on an IPv4-only host and creates an IPv4 listening socket.
- The IPv6 client starts and calls getaddrinfo asking for only IPv6 addresses (it requests the AF_INET6 address family and sets the AI_V4MAPPED flag in its hints structure). Since the IPv4-only server host has only A records, we see from Figure 11.8 that an IPV4- mapped IPv6 address is returned to the client.
- The IPv6 client calls connect with the IPv4-mapped IPv6 address in the IPv6 socket address structure. The kernel detects the mapped address and automatically sends an IPv4 SYN to the server.
- The server responds with an IPv4 SYN/ACK, and the connection is established using IPv4 datagrams.

4. **List and explain the numerous ways to start a daemon.**

i. During system start-up, many daemons are started by the system initialization scripts. These scripts are often in the directory /etc or in a directory whose name begins with /etc/rc, but their location and contents are implementation-dependent. Daemons started by these scripts begin with superuser privileges.

ii. A few network servers are often started from these scripts: the inetd superserver (covered later in this chapter), a Web server, and a mail server (often sendmail). The syslogd daemon that we will describe in Section 13.2 is normally started by one of these scripts.

iii. Many network servers are started by the inetd superserver. inetd itself is started from one of the scripts in Step 1. inetd listens for network requests (Telnet, FTP, etc.), and when a request arrives, it invokes the actual server (Telnet server, FTP server, etc.).

iv. The execution of programs on a regular basis is performed by the cron daemon, and programs that it invokes run as daemons. The cron daemon itself is started in Step 1 during system startup.

v. The execution of a program at one time in the future is specified by the at command. The cron daemon normally initiates these programs when their time arrives, so these programs run as daemons.

vi. Daemons can be started from user terminals, either in the foreground or in the background. This is often done when testing a daemon, or restarting a daemon that was terminated for some reason.

5. **List and explain the actions on startup for syslogd Daemon.**

i. The configuration file, normally /etc/syslog.conf, is read, specifying what to do with each type of log message that the daemon can receive. These messages can be appended to a file (a special case of which is the file /dev/console, which writes the message to the console), written to a specific user (if that user is logged in), or forwarded to the syslogd daemon on another host.

ii. A Unix domain socket is created and bound to the pathname /var/run/log (/dev/log on some systems).

iii. A UDP socket is created and bound to port 514 (the syslog service).

iv. The pathname /dev/klog is opened. Any error messages from within the kernel appear as input on this device.

6. With a function prototype explain the syslog Function.

```
#include <syslog.h>
void syslog(int priority, const char *msg, ...);
```

The priority argument is a combination of a level and a facility, which we show in Figures 13.1 and 13.2.

Figure 13.1. *level* of log messages.

level	Value	Description
LOG_EMERG	0	System is unusable (highest priority)
LOG_ALERT	1	Action must be taken immediately
LOG_CRIT	2	Critical conditions
LOG_ERR	3	Error conditions
LOG_WARNING	4	Warning conditions
LOG_NOTICE	5	Normal but significant condition (default)
LOG_INFO	6	Informational
LOG_DEBUG	7	Debug-level messages (lowest priority)

Figure 13.2. *facility* of log messages.

facility	Description
LOG_AUTH	Security/authorization messages
LOG_AUTHPRIV	Security/authorization messages (private)
LOG_CRON	`cron` daemon
LOG_DAEMON	System daemons
LOG_FTP	FTP daemon
LOG_KERN	Kernel messages
LOG_LOCAL0	Local use
LOG_LOCAL1	Local use
LOG_LOCAL2	Local use
LOG_LOCAL3	Local use
LOG_LOCAL4	Local use
LOG_LOCAL5	Local use
LOG_LOCAL6	Local use
LOG_LOCAL7	Local use
LOG_LPR	Line printer system
LOG_MAIL	Mail system
LOG_NEWS	Network news system
LOG_SYSLOG	Messages generated internally by `syslogd`
LOG_USER	Random user-level messages (default)
LOG_UUCP	UUCP system

- The message is like a format string to printf, with the addition of a %m specification, which is replaced with the error message corresponding to the current value of errno. A newline can appear at the end of the message, but is not mandatory.
- Log messages have a level between 0 and 7, which we show in Figure 13.1. These are ordered values. If no level is specified by the sender, LOG_NOTICE is the default.
- Log messages also contain a facility to identify the type of process sending the message. We show the different values in Figure 13.2. If no facility is specified, LOG_USER is the default.
- The purpose of facility and level is to allow all messages from a given facility to be handled the same in the /etc/syslog.conf file, or to allow all messages of a given level to be handled the same.

7. Explain the significance of daemon_init function.

Figure 13.4 shows a function named daemon_init that we can call (normally from a server) to daemonize the process. This function should be suitable for use on all variants of Unix, but some offer a C library function called daemon that provides similar features. BSD offers the daemon function, as does Linux.

Figure 13.4 `daemon_init` function: daemonizes the process.

daemon_init.c

```
 1 #include    "unp.h"
 2 #include    <syslog.h>

 3 #define MAXFD   64

 4 extern int daemon_proc;         /* defined in error.c */

 5 int
 6 daemon_init(const char *pname, int facility)
 7 {
 8     int     i;
 9     pid_t     pid;

10     if ( (pid = Fork()) < 0)
11         return (-1);
12     else if (pid)
13         _exit(0);               /* parent terminates */

14     /* child 1 continues... */

15     if (setsid() < 0)           /* become session leader */
16         return (-1);

17     Signal(SIGHUP, SIG_IGN);
18     if ( (pid = Fork()) < 0)
19         return (-1);
20     else if (pid)
21         _exit(0);               /* child 1 terminates */

22     /* child 2 continues... */

23     daemon_proc = 1;            /* for err_XXX() functions */

24     chdir("/");                 /* change working directory */

25     /* close off file descriptors */
26     for (i = 0; i < MAXFD; i++)
27         close(i);

28     /* redirect stdin, stdout, and stderr to /dev/null */
29     open("/dev/null", O_RDONLY);
30     open("/dev/null", O_RDWR);
31     open("/dev/null", O_RDWR);

32     openlog(pname, LOG_PID, facility);

33     return (0);                 /* success */
34 }
```

8. **List and explain the steps performed by of inetd Daemon.**

i. On startup, it reads the /etc/inetd.conf file and creates a socket of the appropriate type (stream or datagram) for all the services specified in the file. The maximum number of servers that inetd can handle depends on the maximum number of descriptors that inetd can create. Each new socket is added to a descriptor set that will be used in a call to select.

ii. bind is called for the socket, specifying the port for the server and the wildcard IP address. This TCP or UDP port number is obtained by calling getservbyname with the service-name and protocol fields from the configuration file as arguments.

iii. For TCP sockets, listen is called so that incoming connection requests are accepted. This step is not done for datagram sockets.

iv. After all the sockets are created, select is called to wait for any of the sockets to become readable. Recall from Section 6.3 that a listening TCP socket becomes readable when a new connection is ready to be accepted and a UDP socket becomes readable when a datagram arrives. inetd spends most of its time blocked in this call to select, waiting for a socket to be readable.

v. When select returns that a socket is readable, if the socket is a TCP socket and the nowait flag is given, accept is called to accept the new connection.

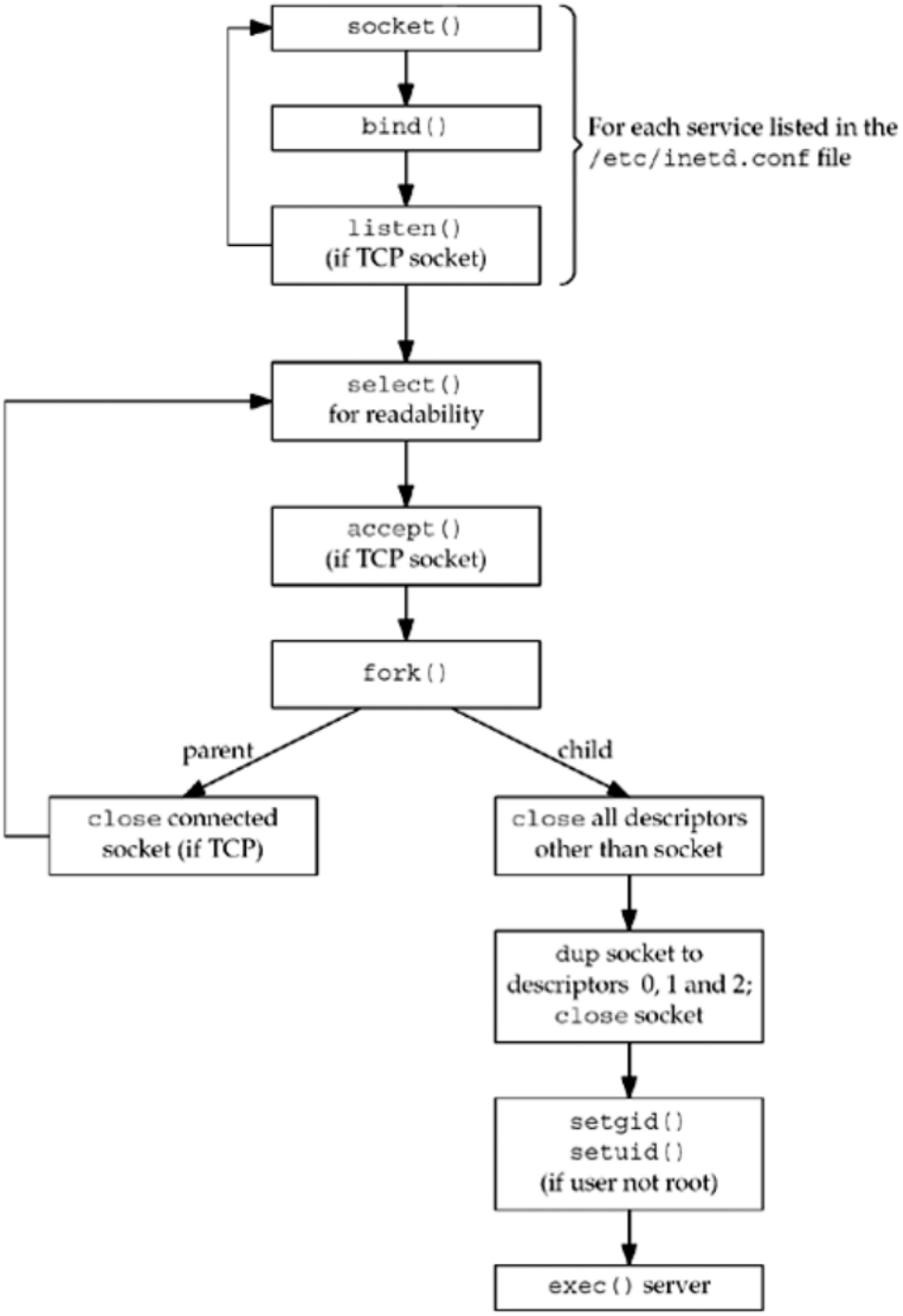
vi. The inetd daemon forks and the child process handles the service request. This is similar to a standard concurrent server (Section 4.8).

The child closes all descriptors except the socket descriptor it is handling: the new connected socket returned by accept for a TCP server or the original UDP socket. The child calls dup2 three times, duplicating the socket onto descriptors 0, 1, and 2 (standard input, standard output, and standard error). The original socket descriptor is then closed. By doing this, the only descriptors that are open in the child are 0, 1, and 2. If the child reads from standard input, it is reading from the socket and anything it writes to standard output or standard error is written to the socket. The child calls getpwnam to get the password file entry for the login-name specified in the configuration file. If this field is not root, then the child becomes the specified user by executing the setgid and setuid function calls. (Since the inetd process is executing with a user ID of 0, the child process inherits this user ID across the fork, and is able to become any user that it chooses.)

The child process now does an exec to execute the appropriate server-program to handle the request, passing the arguments specified in the configuration file.

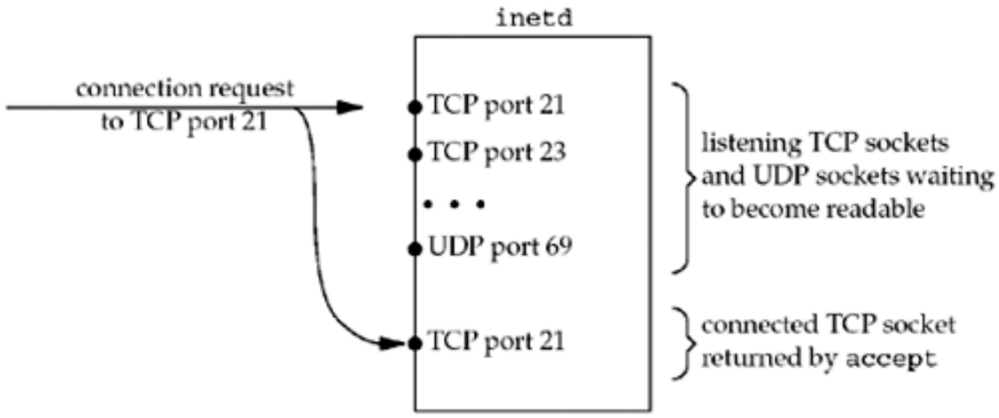
vii. If the socket is a stream socket, the parent process must close the connected socket (like our standard concurrent server). The parent calls select again, waiting for the next socket to become readable.

Figure 13.7. Steps performed by `inetd`.



9. **With a neat block diagram explain the inetd descriptors when connection request arrives for TCP port.**

Figure 13.8. `inetd` descriptors when connection request arrives for TCP port 21.



10. Explain the following with a neat block diagram:

a. inetd descriptors in child

b. inetd descriptors after dup2

Figure 13.9. `inetd` descriptors in child.

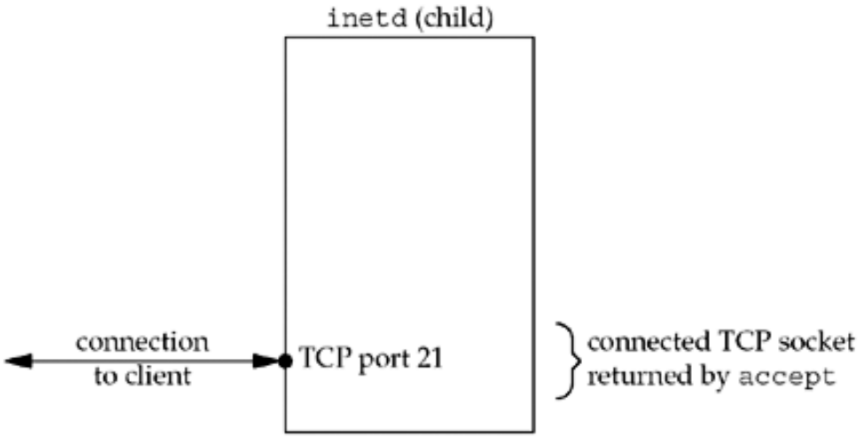
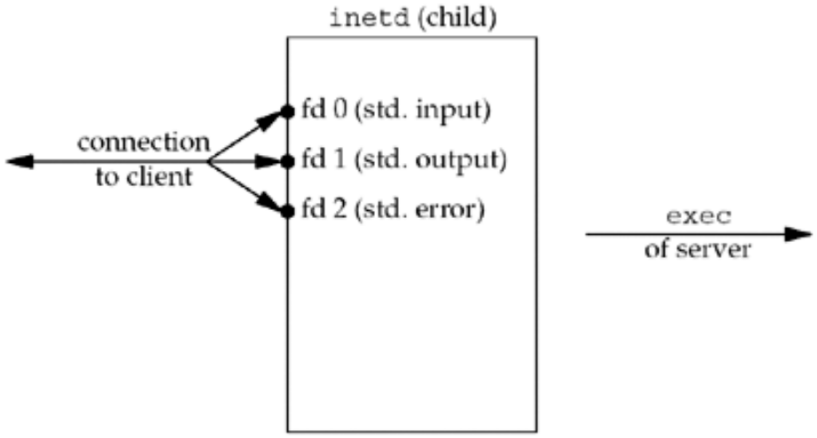


Figure 13.10. `inetd` descriptors after `dup2`.



11. Develop the pseudocode for daemon_inetd function to daemonize process run by inetd.

Figure 13.11 `daemon_inetd` function: daemonizes process run by `inetd`.

daemon_inetd.c

```
1 #include     "unp.h"
2 #include     <syslog.h>

3 extern int daemon_proc;         /* defined in error.c */

4 void
5 daemon_inetd(const char *pname, int facility)
6 {
7     daemon_proc = 1;            /* for our err_XXX() functions */
8     openlog(pname, LOG_PID, facility);
9 }
```